A FLASH TIER FOR TIERED TRANSACTION STORAGE SYSTEM

A Thesis
Presented to
The Academic Faculty

By

Wonhee Cho

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science, College of Computing

Georgia Institute of Technology

May 2020

Copyright © Wonhee Cho 2020

A FLASH TIER FOR TIERED TRANSACTION STORAGE SYSTEM

Approved by:

Dr. Ramachandran, Umakishore
Committee Chair
School of Computer Science
Georgia Institute of Technology

Dr. Ramachandran, Umakishore
Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Sengupta, Sudipta

Amazon.com, Inc.

Dr. Qureshi, Moinuddin
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Pande, Santosh
School of Computer Science
Georgia Institute of Technology

Dr. Jung, Myoungsoo
School of Electrical Engineering
Korea Advanced Institute of Science and Technology

Date Approved: April 21, 2020

BLANK PAGE

To my wife, Juhyo

who decided to dance with me even without knowing much,

and who has been a wonderful partner in my life journey.

Also, to my children, Hansol, Eunsol, and Hayeon

who have been a source of joy and love.

ACKNOWLEDGEMENTS

First of all, I would like to give my utmost thanks to God for His grace and love during this journey. He has always been with me and continues to guide me wherever I go. Even now, regardless of my weakness, I know that He will always give me strength to continue to live my life following Him and graciously lead me.

I wish to express my sincere gratitude to my advisor, Dr. Umakishore Ramachandran, for his guidance, patience, and encouragement during this long journey. His gentle and consistent feedback on this research helped me follow this path to the end.

I am also grateful to Dr. Sudipta Sengupta and Dr. Knut Risvik for introducing me to exciting research areas which led me to this dissertation. It has been a great joy to work with them and explore the cutting edge technology. Additionally, I would like to thank the FaRM team in Microsoft: Dr. Chiranjeeb Buragohain, Dr. Miguel Castro, Dr. Matthew Renzelmann, Timothy Tan, Dr. Shuheng Zheng, Richendra Khanna, Alexander Shamis, and Dr. Aleksandar Dragojevic. It's been a tremendous journey and honor to work with you.

I would also like to thank my remaining committee members, Dr. Moinuddin Qureshi, Dr. Santosh Pande, and Dr. Myoungsoo Jung, for their guidance during the defense, which has helped me in completing this dissertation.

I sincerely appreciate my friends, Yulwon Cho and Dr. Changhee Jung, for the endless conversations, their prayers, reminding me of God in this world, and for the friendship during this journey. Our conversations have always been joyful and refreshing and will undoubtedly continue. I must also thank Yeonju Jeong, Dr. Dushmanta Mohapatra, Enrique Saurez, Dr. Dave Lillethun, Dr. Moonkyung Ryu, Dr. Kirak Hong, Dr. Lateef Yusuf, Harshit Gupta, and Zhuangdi Xu for being wonderful colleagues during this journey.

I am also thankful to my mother, Deokim Oh, who consistently believes and encourages me. Most of all, she has never ceased praying for me and sharing the Word of God.

To my lovely children, Hansol, Eunsol, and Hayeon: I am so grateful for the happiness and companionship you give me. You have grown up so quickly, but I will keep my promise to play with you more often from now on.

Last but not least, I must acknowledge Juhyo, my love and my other half. I cannot thank you enough for your love, sacrifice, support, and endurance during this time. Without you, I couldn't have finished. You have my most sincere appreciation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Modern in-memory distributed storage systems equipped with recent hardware technologies offer both high performance *and* a simple programming model for fairly large data-intensive applications. This enables new application scenarios that have been considered impractical before because the performance was poor or the programming model was complex. Therefore, these new types of in-memory distributed storage systems are getting popular in the datacenter industry.

However, these modern in-memory systems alone cannot tackle the ever-increasing demand for *data-beyond-memory-capacity* as is often seen in systems history, because DRAM is still too expensive to host all the *data-intensive* application's workloads in datacenters, and the *digital universe* keeps expanding without limit. Leveraging cheaper storage is a natural approach to address such capacity demand in a cost-effective manner; however, achieving cost effectiveness without compromising the programming model offered by the new in-memory systems is not trivial due to the disparities between in-memory and storage systems.

In the past, a simple programming model, or ease of programming, was often considered less important, and application developers had to deal with incidental complexities. In contrast, today ease of programming is gaining more importance in the datacenter industry because applications are not set in stone but rather keep evolving to adopt the new requirements from application users.

This dissertation explores the system aspects of a tiered storage system consisting of in-memory *and* flash *tiers* that can provide both ease of programming *and* cost effectiveness. In particular, it focuses on preserving the simple programming interfaces supporting transaction provided by FaRM, a novel in-memory distributed storage system, *and* designing efficient flash-based subsystems to provide cost effectiveness.

The main research question to address in the dissertation is as follows:

How should we architect a tiered transactional distributed storage system such that it can support large data demand effectively while preserving the simple programming model offered by in-memory systems?

The dissertation starts with the hypothesis that while the performance characteristics and available operations of DRAM and flash storage are incompatible, sharing compatible status data between two heterogeneous media and implementing an efficient flash-tier subsystem can provide a solution for the problem.

The dissertation begins by exploring the characteristics of flash storage for a memory-compatible tier and examines a set of techniques to solve the question. As a baseline in-memory storage system, FaRM is used; FaRM is an RDMA-based transactional in-memory distributed system that offers both high performance and ease of programming supporting transaction semantics.

The dissertation covers the related research in distributed storage, in-memory storage, and flash storage systems; then it considers several aspects that a flash storage subsystem must have to be compatible with a memory-tier subsystem. Next, it presents a proposed *T*ired *T*ransaction storage system, T2. During the architecture discussion, it explores the design space of object models, and next it describes the techniques to preserve the simple programming interfaces and to provide efficient transaction from the flash-tier subsystem. The paper subsequently investigates a way to virtualize a memory-compatible space as a region instance to support the same programming APIs from two heterogeneous subsystems. It then discusses how to exploit the transaction-aware mapping table and Compare-And-Swap (CAS) operations for transaction state update to preserve the transaction semantics of the FaRM's commit protocol.

Finally, it evaluates the performance of T2 from different perspectives and concludes.

CHAPTER 1
INTRODUCTION

In the past, the available memory, or dynamic random access memory (DRAM) in computer systems has been much smaller than the size of stored data, even for critical parts of the entire data set. Therefore, the major role of memory is an auxiliary data store, such as *buffers* or *caches*, to provide an illusion of cheap and large capacity memory through the memory-storage hierarchy. This approach has effectively improved the overall performance of systems by hiding the storage's high I/O latency and by exploiting the locality of the workload. While applications of the systems and their workloads have been changed over time, the need for cost-effective storage systems has never been decreased. As a result, designing cost-effective storage systems has been a long-recurring theme in the systems and architecture communities.

However, recent technology advances in memory and storage devices have introduced new opportunities for operating system and distributed system designs. The decreased cost of memory and storage devices allows systems to use more capacity for both devices, non-volatility of novel memory simplifies the design of recovery from the power failure, and the Remote Direct Memory Access (RDMA) technology provides a way to efficiently access the remote memory, bypassing the expensive network stack. This set of technology advancements has led to a paradigm shift in designing storage systems, and research communities and industry have started to use memory as a *primary* data store, not just as a conventional *auxiliary* store in the form of buffers and caches.

More specifically, as DRAM technology has advanced, the cost of DRAM per GB has decreased consistently [1]. This has finally created the momentum to implement a new way to use the large capacity of DRAM more radically so that distributed systems can achieve tens of microseconds of low latency in distributed settings and provide simpler

programming models to programmers. Initially, researchers and system designers simply increased the total capacity of DRAM to use it as a large cache [2] to hide high read latency from storage devices as before; then, various in-memory distributed storage systems [3, 4, 5, 6, 7, 8] were proposed, which completely changed the roles of DRAM and storage devices. The role of memory in these unconventional systems is a primary data store and the role of storage devices is an auxiliary data store for backup or durability.

In these newly proposed systems, or *in-memory distributed storage systems*, DRAM is the genuine store for the data, and storage devices are relegated to a mere backup data store, which is utilized only during data recovery. This *in-memory storage approach* of using memory as a primary data store thus substantially improves system performance, resulting in both low latency and high throughput. Furthermore, non-volatile memory technology removes the storage device access completely from the critical path, which was required for durability in conventional systems. This enables simple programming models that support transactional semantics without sacrificing performance; conventional systems were often designed without transaction supports or provided with weak consistency guarantees [9, 6, 10].

Although the in-memory storage approach is widely adopted throughout industry due to the plummeting cost of DRAM and technology advances in memory and storage devices, the industry faces a cost challenge again as the *digital universe* [11] keeps expanding. Specifically, in-memory distributed storage systems alone cannot tackle the ever-increasing demand for enormous data capacity, including both data size and the number of data objects [11, 12]. The in-memory storage approach relies on the assumption that DRAM can hold all the data sets of applications or critical parts of the data set; however, this assumption becomes invalid very quickly because of the digital universe expansion, and we are again facing the capacity challenge to tackle *data-beyond-memory-capacity*.

In addition, even though DRAM may hold entire data sets or critical parts of them, DRAM cost is still relatively more expensive than storage cost. Furthermore, the main-

tenance cost of datacenter servers having larger DRAM aggravates the situation. For example, DRAM in a general datacenter configuration consumes up to 25% of datacenter energy and will increase as more DRAM is used for primary data store [13, 14]. The maintenance cost and energy consumption, as well as the initial cost to purchase DRAM to serve huge data capacity, are the major hurdles to the wide deployment of in-memory distributed storage systems at scale.

Therefore, in-memory distributed storage systems alone are not a practical solution for a datacenter to serve data-intensive applications that require enormous data capacity; furthermore, they cannot keep pace with data growth in the long run [3]. In addition, workloads of internet-scale applications are skewed, and not all data is the same [15, 16] as hot and cold data should be treated differently. Therefore, industry is seeking a cost-effective solution to tackle the capacity challenge without losing the benefits of in-memory storage systems.

Leveraging cheaper storage is a conventional approach to address such capacity demand in a cost-effective manner, and applications are built using two heterogeneous in-memory and storage-based systems. However, application development on these hybrid systems is not trivial in modern datacenters because of the complexity of the systems. For example, mistakenly handling different consistency guarantees can easily result in data loss. Therefore, instead of application developers managing such complexities due to discrepancies, the storage systems as a whole must provide common interfaces and consistency guarantees. In the modern environment, ease of programming is not just an option but a critical requirement [17] because applications are not set in stone but rather keep evolving to continuously adopt new requirements from application users.

This dissertation explores the system aspects of a tiered storage system consisting of in-memory *and* flash tiers that can provide both ease of programming *and* cost effectiveness. In particular, the focus is on preserving the simple programming interfaces that supports transaction semantics provided by an in-memory distributed storage system, *and* designing

an efficient flash-based subsystems to provide cost effectiveness. The dissertation subsequently presents a tiered transaction storage system architecture as a solution.

1.1 Problem Statement and Research Questions

The research question in the dissertation is as follows: How should we architect a tiered transaction storage system, such that it can support large data demand effectively while preserving the simple programming model offered by in-memory systems?

More specifically, this dissertation focuses on the flash-tier subsystem design and its integration with the existing in-memory storage system, such that 1) the flash-tier subsystem preserves the simple programming interfaces to cooperate with the memory-tier subsystem through the common commit protocol and 2) the flash-tier subsystem performs efficient transaction to achieve the best performance of flash storage devices. This means that the performance of the proposed system is comparable to pure memory design in optimal conditions and degrades gracefully as flash I/O is involved.

1.2 Thesis Statement

In-memory transaction storage systems with modern hardware provide high performance and ease of programming for data-intensive applications; however, they are still expensive in terms of supporting large data capacity and are not cost effective in supporting common datacenter workloads. A typical hybrid approach using secondary storage systems alongside the in-memory transaction storage systems may enhance the overall cost effectiveness, but the disparate combination of in-memory transaction storage and secondary storage systems complicates datacenter application logic and compromises the crucial benefit of ease of programming empowered by the in-memory storage systems.

A tiered architecture with a flash-tier subsystem that shares transaction status and provides transaction operations can effectively improve cost effectiveness while preserving ease of programming.

1.3 Contributions

The contributions of this dissertation are the following:

- This dissertation presents a tiered transaction storage system architecture that supports both in-memory and flash storage workloads through the same programming model. This architecture also allows the system to utilize the same commit protocol while isolating the differences of physical mechanisms.
- This dissertation examines the performance characteristics of different storage access mechanisms and considers various object models for the objects in the flash tier to support the tiered storage system.
- This dissertation explores several techniques for the flash-tier subsystem to support system-wide transaction efficiently and to achieve high performance from the underlying flash storage.
- This dissertation evaluates the performance of the flash-tier subsystem quantitatively using two custom micro-benchmarks and YCSB and the ease of use qualitatively. The result shows that T2 can achieve cost effectiveness by providing the competitive or better throughput for the same cost while preserving ease of use for applications.

1.4 Dissertation Outline

The dissertation consists of seven chapters. This chapter presents the problem and thesis statement. Chapter 2 discusses related work on in-memory and flash-based distributed storage systems and background. Chapter 3 considers the performance and cost aspects of storage for a tiered transaction storage system, and Chapter 4 describes the architecture and explores the design space of potential object models for the flash-tier subsystem. Chapter 5 describes the techniques to provide both cost effectiveness and ease of program-

ming. Chapter 6 evaluates the performance of the system. Finally, Chapter 7 concludes the dissertation and presents future work.

CHAPTER 2
BACKGROUND AND RELATED WORK

2.1 Introduction

This chapter first clarifies the terminology used in this dissertaton and explains the context for the topics of the dissertation. It then describes the context of distributed storage systems and briefly explains FaRM [9, 18], the in-memory distributed storage system that we use as a base platform. Next, it introduces related work on tiered storage systems, flash storage systems, and in-memory distributed storage systems. Finally, it explores a new challenge that is introduced by new requirements in the datacenter industry; in sum, a storage system should tackle *data-beyond-memory-capacity* through uncomplicated semantics.

2.2 Terminology: Memory, Flash, Storage Hierarchy, and Tiered Storage System

In this dissertation, the terms "memory and DRAM" are used interchangeably to refer to volatile DRAM technology, and the term "storage devices" is used to refer to traditional hard disk and flash-based technology, excluding DRAM. However, when storage is used in the context of a system to store data, both DRAM and storage devices are considered storage. The term "data store" is used to refer to both technologies as storing media. A solid state drive (SSD) is a durable storage device that uses solid state (NAND flash) technology [19], and its price and performance lie between those of memory and traditional hard disks [20]. The terms "flash storage devices" and "solid state drives" (SSDs) are used interchangeably. Storage hierarchy is one of the foundational concepts in storage systems that has been investigated for more than four decades [21], which puts faster but more expensive storage devices on top of slower but cheaper ones to provide an illusion of both fast access *and* large capacity. A tiered storage system generally refers to a storage system that uses

heterogeneous storage tiers, such as traditional disks, tape, and solid state drives (SSDs) to improve performance and cost effectiveness. However, in this dissertation, the concept is expanded to include memory as another storage tier and also to provide mechanisms for differentiating requests based on different performance needs.

2.3 Context: Storage Technologies, Distributed Storage Systems, and Storage Systems Requirements

The context of the dissertation is a distributed in-memory transaction storage system and its modification through flash technology to expand the capacity of the system. Therefore, this section describes the background of storage technologies, distributed storage systems, and new requirements and challenges in current datacenter environments, setting the context for the topics explored in this dissertation.

2.3.1 Storage Technologies

The advances in DRAM and storage technologies have driven storage system architecture. Although numerous storage systems have provided an illusion of better performance with cheaper cost, the fundamental storage hierarchy has remained the same regardless of whether it is built in operating systems or in distributed systems; more expensive and smaller DRAM lie on top of cheaper and larger storage devices.

Specifically, over the past several decades traditional hard disk drives (HDDs) have been the *de facto* primary data store, and DRAM has served as an auxiliary intermediate data store to provide low latency while achieving cost effectiveness. The most common technique is to use DRAM as a *read cache* and a *write buffer* on top of slower storage devices, or HDDs (Figure 2.1). This storage hierarchy effectively hides high latency due to the physical limitations of storage devices, such as the hard disk's slow seek time. In addition, the *log-structured mechanism* has been commonly employed to mitigate the performance gap between sequential and random writes [22] by converting slow random write

requests into fast sequential write requests. In essence, the common goal of these techniques is to hide the latency of slow storage devices from the data access path in storage systems.

The recent introduction of NAND technology, which is deployed as solid state drives (SSDs), has changed the role of spinning disks as secondary or backup media.

Because a flash-based SSD does not have a moving physical head assembly, it removes the slow seek time and provides much lower latency [23]. Moreover, it also enhances random writes and reads by exploiting smart mapping (Flash Translation Layer) and internal cache techniques.

Initially, SSDs were deployed as a cache over hard disk drives (HDDs) because the cost and performance lie between those of DRAM and HDDs [20]. SSDs can effectively hide the slow performance of HDDs by exploiting the common read cache and write buffer techniques [24].

With the advances in NAND technology, such as Multi-Level Cell (MLC) and Triple-Level Cell (TLC), the cost of SSDs has gone down and has reached a reasonable cost per byte compared to the disk (Figure 3.3). As a result, SSDs have started to take the role of a primary data store in storage systems, replacing HDDs and relegating them to a backup data store.

In the meantime, the cost of DRAM plummets, and industry has started to deploy DRAM extensively to hold most of the critical application data in a DRAM-based cache. For example, many applications, such as social graphs, consist of critical small-size meta data and large-size non-critical non-meta data [2, 25], and the capacity of the DRAM cache is enough to hold critical parts of data and support low-latency operations. However, the role of DRAM still remains as an *auxiliary* data store to improve overall storage performance by hiding the high latency of *primary* storage devices.

The recent debut of in-memory distributed storage systems [3], such as RamCloud [26, 27] and FaRM [9, 18], has introduced a major paradigm shift in the storage hierarchy. In

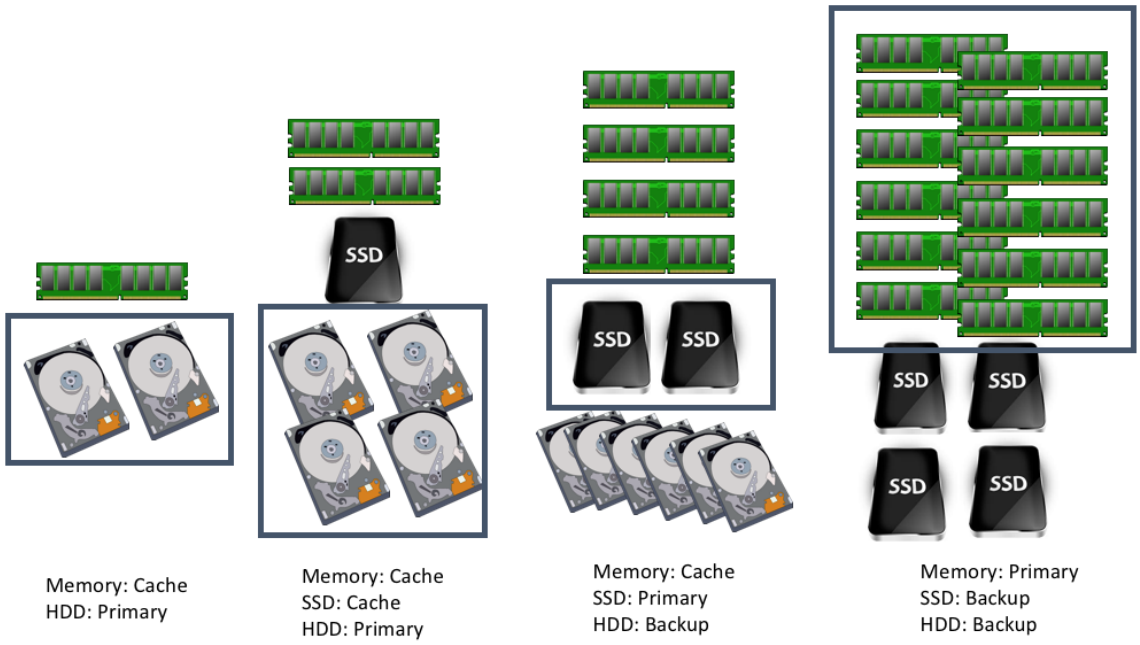


Figure 2.1: Primary Data Store in Storage Systems. While the technology advances in memory and storage devices have improved available capacity and performance over the past decades, the role of memory has remained as an auxiliary data store, or cache, until recently. Recent in-memory distributed storage systems (right-most) introduce DRAM as their primary data store, which makes it possible to provide low latency, high throughput, and a simple programming model in a scalable way.

in-memory distributed storage systems, DRAM itself serves as the primary data store, and storage devices are used for backup data stores for availability and durability. Although the concept of in-memory storage systems has been investigated for several decades [28, 15], it does not consider current challenges in distributed storage systems, such as scalability and availability issues.

Moreover, these memory-only distributed storage systems can exploit various performance technologies such as RDMA [29], which is practically available for memory and achieves additional performance improvement [9, 18]. Removing the storage devices from the critical data path allows the systems to provide not only high performance, but also a simple programming model, such as having transaction semantics, which has often been sacrificed for performance and availability [30]. The implication of these changes in storage hierarchy is that in-memory storage systems are highly optimized for memory performance;

therefore, the main challenge is to design a memory-compatible and high-performance storage tier so that the resulting tiered storage system preserves the benefits of in-memory storage systems while improving capacity with expected performance degradation, depending on the target workload.

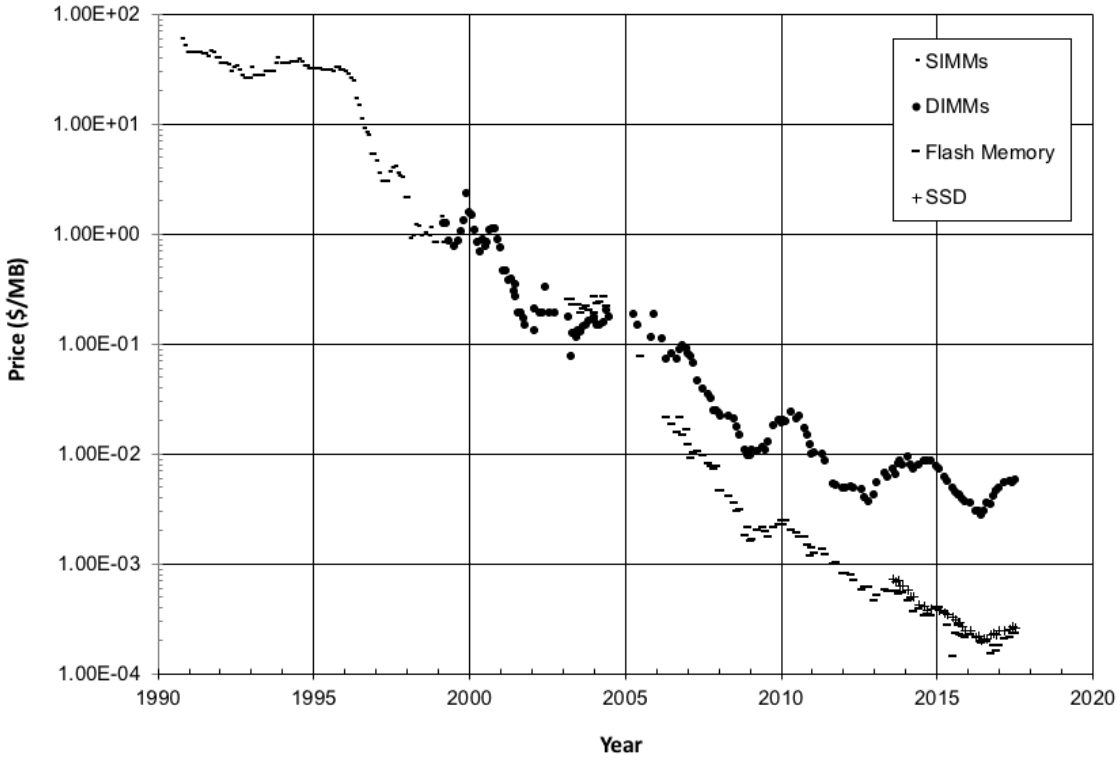


Figure 2.2: Historical Cost of DRAM and Flash-based Storage. A reproduced version of John C. McCallum's historical cost of DRAM and flash memory storage and/or solid state drives [1]. The original price and capacity were collected from publicly available market prices and magazines. Although the DRAM cost per dollar goes down, the flash cost does so more rapidly, and the gap between the two increases relatively over time.

2.3.2 Distributed Storage Systems

Traditionally, large-scale distributed storage systems have shared common goals [31], and the common goals are to provide performance, scalability, consistency, and availability at the minimum [25, 30, 2, 32, 33]. These large-scale systems utilize commodity memory and storage devices, and they span several hundreds to several thousands of multiple

servers [34, 31] (Table 2.1).

Each system has a different emphasis on what to prioritize, and this creates a diverse spectrum of distributed storage systems. As a result, various distributed storage systems [35, 36, 25, 2] have been designed to meet the unique requirements and needs of applications. However, not all the goals can be met, and a trade-off is indispensable. For example, strong consistency is often weakened to achieve low latency and high availability [25, 2, 30]; an application may need extremely low latency and fast interaction [25, 2], but huge storage capacity may not be important, or an application may need massive transactions but with reduced consistency semantics [30].

Regardless, these diverse distributed storage systems still have the common storage hierarchy that uses memory as an auxiliary data store.

2.3.3 Requirements for Distributed Storage Systems for Datacenters

The design and implementation of distributed storage systems are driven by application requirements; however, practical deployment is constrained by cost effectiveness. Therefore, the right design architecture choice is often made by understanding the requirements of applications and expected performance and also exploiting the deployable technologies at hand.

For distributed storage systems used as a datacenter platform to support online data-intensive applications, low latency and high throughput are critical requirements. However, without significant cost reduction, in-memory distributed storage systems alone cannot be deployed as a datacenter platform. Although cheap DRAM cost makes large-cache systems [2, 34] and even in-memory only distributed storage systems [27, 26, 9] viable, the most critical factor that prevents the practical deployment of in-memory distributed storage systems as a datacenter platform is still the high cost per GB for serving data.

Additionally, we observe that huge data demand *beyond the memory capacity* is on the horizon. The size of the data universe exponentially increases over time via consumers and

enterprise digitization [11]. For example, a single data object–such as document files and multimedia files– becomes more complex and larger. Consumers generate huge amounts of content, and industry digitizes its existing content, including media, health care, video surveillance, etc. Moreover, the Internet of Things (IoT) accelerates this trend by generating continuous sensor data in various sizes and formats.

Therefore, the promising DRAM cost reduction cannot match the speed of data generation, not just for the overall data spectrum, but even the size of the critical data part is also increasing, excluding data objects that can be processed offline through batch processing. For example, interactive applications, such as social graphs [25, 2], knowledge graphs, and recent AI-based virtual assistants access and manipulate huge amounts of data online, which requires low latency from their inception.

In addition, the development of such applications also requires a simple programming model to manage data [37, 38]. The importance of this aspect has recently increased because, in a modern datacenter development setting, thousands of developers access application code and significant software is often released multiple times per week [17].

One opportunity in this trend is that not all the data have the same value. In the recent literature, we find that many online transaction processing (OLTP) and online serving traces are skewed [15, 15]. In addition, many industry application developers design their application data structures to differentiate critical *meta data* from non-critical data [25]. For example, they keep the critical data in low-latency DRAM-based cache and large-size non-critical multimedia data in storage devices, and they differentiate the performance of these different data types.

In this dissertation, we seek a solution to meet these requirements for *data-beyond-dram-capacity* and a simple programming model on top of an in-memory distributed storage system. We use FaRM [9] as a base in-memory distributed storage system, which already provides low latency and high throughput, as well as a simple programming model that supports transanction.

2.4 Background: FaRM

FaRM [9] is the in-memory distributed storage system that this dissertation uses as a base in-memory system. This section explains the terminology, such as region and object, programming model, and the transaction-replication protocol of FaRM [9, 18], which are related to a flash-tier subsystem design.

FaRM is a main memory distributed storage system that provides low latency and high throughput by exploiting recent hardware advances, such as RDMA and non-volatile memory. It bypasses the heavy kernel stack and accesses remote memory through RDMA for data access and message passing. FaRM provides a simple event-driven programming model and supports strictly serializable ACID transactions and high availability through replicated logging. Moreover, FaRM achieves greater performance improvement through one-sided RDMA, lock-free read-only operations, and co-location.

2.4.1 Data and Programming Model

FaRM exposes cluster-wide shared memory via a globally shared address space to applications, which manages a data object through simple application programming interfaces (APIs) that support transaction. The supporting operations are *txstart*, *alloc*, *read*, *write*, and *txcommit* (Figure 2.3).

Internally, FaRM divides the shared memory by a region of 2GB, which is a unit of registration for RDMA-management and recovery. Each region is grouped by 1 MB blocks, and then each block is divided into slabs having several levels of sizes. When an application requests an object, FaRM allocates an object from a specific region and returns a 32-bit address to the application. The address consists of a region identifier and its internal offset; however, it is opaque to the application.

When an application thread starts a transaction, it becomes a transaction's *coordinator*. The coordinator performs arbitrary data management operations such as allocation,

```
Tx* txCreate();
void txAlloc(Tx *tx, int size, Addr addr, Cont *c);
void txFree(Tx *tx, Addr addr, Cont *c);
void txRead(Tx *t, Addr a, int size, Cont *c);
void txWrite(Tx *t, ObjBuf *old, ObjBuf *new);
void txCommit(Tx *t, Cont *c);
```

Figure 2.3: FaRM's API. FaRM [9] provides simple memory management and transaction primitives. *txCreate* creates a transaction context for the following operations. *txAlloc* allocates an object from the shared memory space and *txFree* frees an object. *txRead* reads object data from the shared space. However, *txWrite* updates an object in its local heap first, and its update is applied to the shared memory space only when *txCommit* completes successfully

read, and write, and manages the transaction. When an application commits the transaction, it results in success or abort, depending on its concurrency context or conflict. Only successfully committed objects are exposed globally.

2.4.2 Transaction-Replication Protocol

FaRM provides availability by replicating objects in its backup servers. When f is the fault tolerance for the system, a region has one primary server and f backup servers to replicate the region. In a normal state, when an object is read, it is read only from its primary region; when an object is written, it is committed to both primary and backup regions after a successful transaction. The backup region can be stored in memory or in flash storage.

FaRM's commit protocol integrates both transaction and replication for performance improvement [9, 18]. A transaction has two phases: execute and commit. During the execute phase, an application accesses several objects and may update some of the object data. When an application issues a commit operation for a transaction, FaRM starts a five-stage commit protocol: lock, validate, commit backup, commit primary, and truncate. During the commit protocol, a coordinator, which runs the transaction, sends and receives messages for these stages and records the logs containing the object information, such as address and version. The logs are stored in non-volatile ring buffers in each server for performance and durability, and then they are applied in their address space. When a data

object is remote, FaRM uses one-sided RDMA operations; however, when a data object is local, it uses local memory accesses.

2.5 Related Work in Tiered Storage Systems

A tiered storage system uses hybrid storage devices to achieve cost effectiveness by exploiting the storage hierarchy inside the system. Traditionally, it uses different storage devices, including tapes, hard disks, and solid state drives, and the tiered system exposes itself as a single logical storage system while providing the higher tier's performance in the optimal case, where most target workloads can be processed from the higher tier.

The two commonly adopted techniques in the tiered storage system are 1) to use the smaller but faster tier as write buffers to hide write latency by keeping data in the buffer and lazily flushing them later and 2) to use it as read caches to keep the recently read data in the cache or even prefetch data from the slower tier to reduce read latency when the data are requested. The size of write buffers is generally smaller than that of read caches, and the performance improvement by read caches depends on the target workload's access pattern and caching policy. Traditionally, memory is used for those buffers and caches, but with the adoption of flash technology SSDs are used for the buffers and caches and often are incorporated on top of traditional hard disks as an intermediate tier [24, 39].

T2 also exploits the two techniques for flash-tier performance improvement; moreover, it addresses the need to support transactions in the context of distributed storage systems, which is often not addressed in the traditional setting. Some flash storage systems also support transaction mechanisms in the context of distributed storage systems [40, 41]; however, those transaction systems are designed to provide separate flash-based transaction operations. In contrast, T2 is integrated into in-memory distributed storage systems to achieve efficient transactions.

Table 2.1: Distributed, In-Memory Distributed, and Flash Storage Systems.

Distributed Storage Systems	**In-Memory Distributed Storage Systems**
Dynamo (2007)	RamCloud (2011, 2015)
Redis (2008)	FaRM (2014, 2015)
Memcache, TAO (2013)	MICA (2014)
Flash-based Storage Systems	**Hybrid Approaches**
FAWN (2009)	Cache + File system
FlashStore (2010)	In-Memory + Flash
SkimpyStash (2013)	Cache + Flash
LLAMA (2013)	...
Deuteronomy (2013)	

Distributed storage systems provide performance, availability, and scalability by default and often use a large-size DRAM cache for performance improvement. In-memory systems use DRAM as their primary data store to further improve performance and even stronger consistency. The systems in the flash storage systems section focus on the characteristics and performance of flash storage devices. Hybrid approaches represent a combination of two separate systems, which is commonly taken by application developers to resolve the capacity issue.

2.6 Related Work in Flash Storage Systems

Solid state drives (SSDs) [19] have been actively investigated and widely deployed in industry because of their unique characteristics including performance, energy advantages over those of spinning disks [23, 42, 43], and attractive cost performance, which resides between that of DRAM and hard disks [20, 1]. SSDs have three unique characteristics compared to traditional hard disks: first, they use NAND-flash technology to store data; there are no mechanical moving parts, which completely removes the seek time to access data. This physical characteristic gives much lower I/O latency. Second, the units of erase and write operations are different, and the block of cells to be written should be erased first. Finally, the memory cells have a limited lifetime and wear out after a certain number of erase operations. Because of the last two characteristics, SSDs internally exploit remapping, caching, and garbage-collection techniques. These techniques differentiate the performance of random vs. sequential operations and read vs. write operations.

Distributed storage systems also exploit SSDs to utilize higher performance than HDDs with cheaper cost than DRAM. Although the I/O performance of SSDs is much higher

than that of HDDs, there is a gap between random and write operations, so that distributed storage systems commonly exploit log-structure techniques [32, 40, 44, 22] to improve write performance. LLAMA [40] is a flash-based storage system that is close to the flash tier in T2. LLAMA uses flash storage to provide both cache and storage management and to support transactions. It uses *states* in the page-oriented mapping table, uses latch-free compare-and-swap (CAS) operations to support the transactions, and employs write buffers, or flush buffers, to mitigate the latency resulting from write operations to SSDs. T2 takes the LLAMA approach regarding using states, CAS, and flush buffers to support transactions and fast write operations. However, T2 is designed to work with its memory tier and integrates the higher transaction logic inside a transaction-aware mapping table for efficiency and simple semantics. Instead of implementing separate transactional logic, T2 uses the same transaction-and-recovery protocols of FaRM [9] so that transaction operations are simplified.

2.7 Related Work in In-Memory Distributed Storage Systems

Recent advances in memory technology [45] have accelerated the adoption of these memory-based distributed storage systems [26, 27, 46, 47, 9]. As a result, researchers started to use memory as primary data store, which is often called an *in-memory data management system* or *in-memory storage system*. An in-memory storage system relegates traditional storage devices to a secondary data store only for durability and availability. By taking high-overhead I/O devices out of datacenter application logic, in-memory distributed storage systems achieve extremely low latency and high throughput, up to 100-1000x better than those of disk-based systems [27]. FaRM [18, 9] and RamCloud [27, 7, 26] are two representative in-memory storage systems which target large-scale in-memory data storage and show low latency and high throughput. While RamCloud provides simple key-value operations, FaRM provides memory management operations and also supports transactions. The latency is dramatically reduced; storage is no longer a hindrance to access latency, and

it provides a simple programming model for application developers [48, 9].

Nonetheless, these approaches do not provide an answer for the *data-beyond-memory-capacity* requirements needed in the big-data era [11]. T2's approach complements the missing feature of in-memory distributed storage systems rather than competing with the in-memory storage systems. Specifically, our goal is to achieve the cost effectiveness *and* simple programming model that are already provided by in-memory distributed storage systems.

2.8 Related Work in *Data-Beyond-Memory-Capacity*

Recently, several approaches have been used to resolve the larger-than-memory issue [49, 50]; however, the resolutions are limited because they do not address datacenter application requirements.

Traditionally, developers in the datacenter industry have resolved this capacity gap through a hybrid approach. The in-memory distributed storage systems are used for critical data paths and a separate storage-based distributed storage system, such as a flash-based key-value store or a file system, can be used to handle data capacity that does not fit in the available memory. However, in this approach programmers should be able to handle different semantics of using different media-oriented systems; for example, when an application needs to address larger data, application developers manually implement separate logic to handle these two media operations. But exploiting two separate storage systems with different semantics is not easy for developers; this usually results in unnecessary complexities in application logic as well as inefficiency at best, and bugs in the programs at worst. In modern datacenter application development, thousands of application developers are accessing or updating application code, and the applications should be shipped frequently, that is, several times per week[17]. These frequent development cycles force the datacenter industry to seek simple programming models that can handle such unnecessary complexities [17, 48, 9].

Our approach (Table 2.2) is similar to hybrid approaches in the sense that we present storage devices as the primary data store alongside the memory tier; however, we also present it such that it can be accessed and processed through the same programming model as a current in-memory system can.

Table 2.2: Comparison of Representative Cache-based, In-memory, and Flash-based Distributed Storage Systems.

Name	Memcache	RamCloud	FaRM	LLAMA	Hybrid*	T2
Approach	Cache	In-Memory	In-Memory	Flash	Hybrid	Hybrid
Primary Media	Storage	DRAM	DRAM	Flash	Storage	Memory + Storage
Programming Model	KV Store	KV Store	Shared Memory	Page	KV Store + Files	Shared Memory
Capacity (> Memory)	V	-	-	V	V	V
Transaction Support	-	-	V	V	-	V
Performance	++	+++	+++	+	++/+	+++/+

The Name row lists representatives of each approach, except Hybrid, which is a general approach taken by application developers to use any of cache-based/in-memory distributed systems or flash-based systems. V (check) and - (dash) represent whether a system supports (V) or not (-). The number of + (pluses) represents the relative but not scaled performance of each system. / (slash) differentiates the performance between two primary data stores.

2.9 Summary

Recent distributed storage systems started to use memory as a large cache and even as their primary data store, and prior work focused on in-memory systems, assuming that most or a critical part of data can be stored in memory. However, recent data expansion has created critical challenges to such approaches because data demands are beyond the practically and economically available memory capacity.

To support the *data-beyond-memory-capacity*, a current practical solution for the datacenter is to take a hybrid approach of using two separate distributed storage systems; however, this results in unnecessary complexities for reasoning two independent transaction semantics and inefficiency for application development. Instead, distributed storage systems should handle such complexities in such a way as to elicit the benefits of memory, such as low latency, and the benefits of large capacity with simple semantics and ease of programming.

CHAPTER 3

STORAGE CONSIDERATION FOR TIERED SYSTEM

This chapter begins by describing the design goals and principles of the proposed system. Next we discuss the implication of using secondary storage as a memory-comparable tier. We first evaluate the system cost of tiered storage systems and then address the requirements for the secondary storage subsystem from two perspectives: access methods and efficient transaction support. Finally, we explore the performance characteristics of file access methods on Windows systems in different configurations.

3.1 Design Goals and Memory Compatibility

3.1.1 Design Goals

The two primary goals of the proposed tiered transaction system are to preserve ease of programming provided by the existing in-memory system, FaRM *and* to make the overall system cost effective. One preference in the design is to provide more high-performance memory to users than to provide high-performance flash space by utilizing the memory for cache.

Cost *effectiveness* usually implies that the overall system provides an illusion of the expensive media's high performance with the larger but low-performance media. However, *effectiveness* in this dissertation does not mean achieving the performance of memory tier from the enlarged proposed tiered transaction system. Rather, it means achieving the best performance of flash storage devices and minimizing the overhead of supporting the memory-compatible transaction. It is possible to provide such a high-performance illusion if the system uses *enough* auxiliary memory to cover a common workload. However, this approach results in less memory for the memory tier. The target system must provide most

of the available system memory as the memory tier and use as little memory as possible for the auxiliary purpose.

3.1.2 Memory Compatibility

The programming model of FaRM offers globally shared *memory* spanning multiple servers through a set of APIs. An object in this memory is identified by a handle, or an *address*, and objects are manipulated via *low-level access methods*, such as *Alloc*, *Read*, *Write*, *Delete*, etc. Such data operations are performed in the context of *transaction* (section 3.3). In this dissertation, we use the term *memory-compatibility* to refer to a storage system that conforms to this programming model.

Most storage systems built with commodity storage devices are usually not compatible with this transactional memory model. Several flash-based storage systems, such as LLAMA [40], provide transaction APIs, and several transactional flash storage devices are also proposed [51, 52]. There are still no available commodity storage devices implementing the proposed flash-level transaction, and LLAMA's caching/storage systems are not designed for the tiered architecture. However, LLAMA's approach of achieving the high performance can be applicable to the flash-tier subsystem design.

In the following sections, we examine storage from two perspectives: cost effectiveness and memory comparability.

3.2 Tiered Storage System Cost

The primary driving force in pursuing a tiered system is to reduce the overall cost of storage systems and to increase capacity per dollar. Although the plummeting cost of DRAM offers various in-memory high-performance storage systems, *relative* cost effectiveness always has been the major factor in building systems in a datacenter [53] or designing internet-scale applications. While the available capacity of DRAM in newer-generation stock-keeping units (SKUs) in datacenters increases, the demand for more capacity or lower

cost has never declined, and system designers seek cost-effective solutions with minimal performance impact.

3.2.1 Storage Configuration and Cost

A typical datacenter server is equipped with memory, HDDs, and flash disks that are configured to achieve cost effectiveness [14] with minimal performance impact. Although the actual configuration of memory and storage devices, including both HDDs and flash disks, varies in each generation of SKUs, we can estimate the relative capacity ratio of DRAM and storage devices. For example, with the full 52U rack with 96 servers in a typical server configuration (Figure 3.1), the maximum capacity of flash disks can be 16 times greater than that of DRAM.

In this section, we use this configuration to discuss the overall cost effectiveness of the proposed system, which adopts a flash disk as a comparable tier to an existing in-memory storage system. Although HDDs can additionally reduce the cost and further increase available data capacity, the performance of HDDs for I/O operations is inferior to that of SSDs (Table 3.2). Therefore, this dissertation will be limited to considering a flash-based tiered structure.

Table 3.2: Memory and Storage Cost.

Type	**Model**	**Dollar / GB**	**Sequential**	**4K Read**
Memory	DRAM DDR3 8GB 1333/1600MHz	7.13		
SSD	512GB-VNAND SATA III	0.31	520 MB/s	36.6 MB/s
HDD	1TB-SATA 6 Gb/s 7200 RPM	0.05	180MB/s	1.31 MB/s

Prices of commodity DRAM, SSD, and HDD on September 29, 2017. Although the DRAM cost has come down over the decades, it is still 23 times more expensive than SSD. The cost of HDD is six times cheaper than that of SSD. The performance in the table is based on the user reports in UserBenchmark [55], not from the vendor specification.

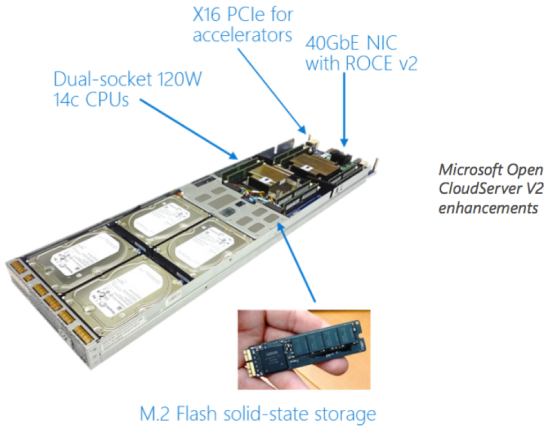


Table 3.1: Server Configuration.

Technology	Full Specification
Memory	48 TB
Flash (SSD M.2, PCI-E)	3/4 PB
HDD	2.3 PB
RDMA	3.8 Tbps

Figure 3.1: Datacenter Server Configuration. With the full 52U rack with 96 servers that Open CloudServer V2 (October 2014) provides [54], flash capacity is 16 times greater than that of memory.

3.2.2 Proposed Storage Configuration

FaRM [9] replicates data on one primary replica and f backup replicas to provide availability during f server failures. In the three-replica case, it can be configured to store all the data on three memory replicas or to store on one primary memory replica and two backup SSD replicas. The latter approach reduces the cost by utilizing cheaper, durable SSDs; however the cost is still high. The proposed tiered storage system increases data capacity even further by adopting SSDs as a primary replica as well as backup replicas. Figure 3.2 shows the storage layout for the proposed system with the typical datacenter configurations (Figure 3.1). The proposed system can effectively increase the available data capacity up to seven times compared to the second approach in the maximum configuration.

As a result, the relative cost to provide the same data capacity with a tiered architecture decreases as the total user-available data capacity increases. Figure 3.3 shows the relative cost of the system. For simplicity, this graph ignores other cost factors, such as maintenance, CPU, network cards, etc. When baseline FaRM utilizes DRAM for primary data and SSDs for two backup replicas, we regard the *relative data capacity* of this configu-

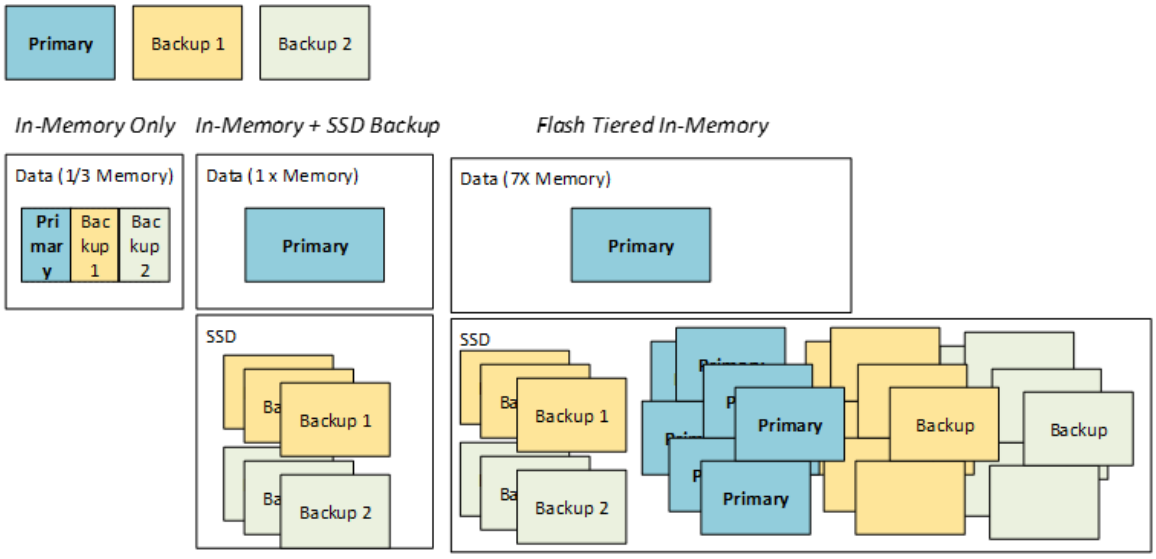


Figure 3.2: Relative Available Capacity. Total available relative capacity for the given DRAM capacity (1) for three different approaches: 1) three memory replicas, 2) one primary memory replica and two backup SSD replicas, and 3) the second approach + one primary SSD replica and two backup SSD replicas.

ration as one which is available to users. Then, we increase the relative data capacity by adding more DRAM and/or SSDs in two system configurations. FaRM adds more DRAM and SSDs (blue), and T2 (orange) adds SSDs for primary and backup replicas (orange). The overall relative system cost drops dramatically when we expand the relative data capacity by adding SSDs for primary and backup replicas. For example, when we provide four times the baseline data capacity, the additional system cost is three times the baseline price; in contrast, T2 just needs an additional 50% of the baseline cost to provide the same data capacity through SSDs because the price of DRAM is 23 times higher than that of SSDs (Table 3.2).

In actual scenarios, more DRAM not only increases the initial cost for DRAM, but also amplifies its maintenance cost because of DRAM's high power consumption. Moreover, each server allows only a limited number for DRAM slots; the purchase of additional servers is indispensable for increasing the DRAM capacity. Therefore, the relative cost difference between the two systems increases more than that of our estimation.

3.3 Storage Subsystem Primitives

Leveraging cheaper storage is a natural step to achieve cost effectiveness. However, achieving it without complicating the programming model provided by the in-memory systems is not trivial. Specifically, the disparities between DRAM and flash storage pose a major challenge to the system design.

In conventional hybrid system approaches (section 2.8), application developers choose a separate in-memory system and storage system and utilize them in an application. However, each system is not designed in the same context and has a different programming

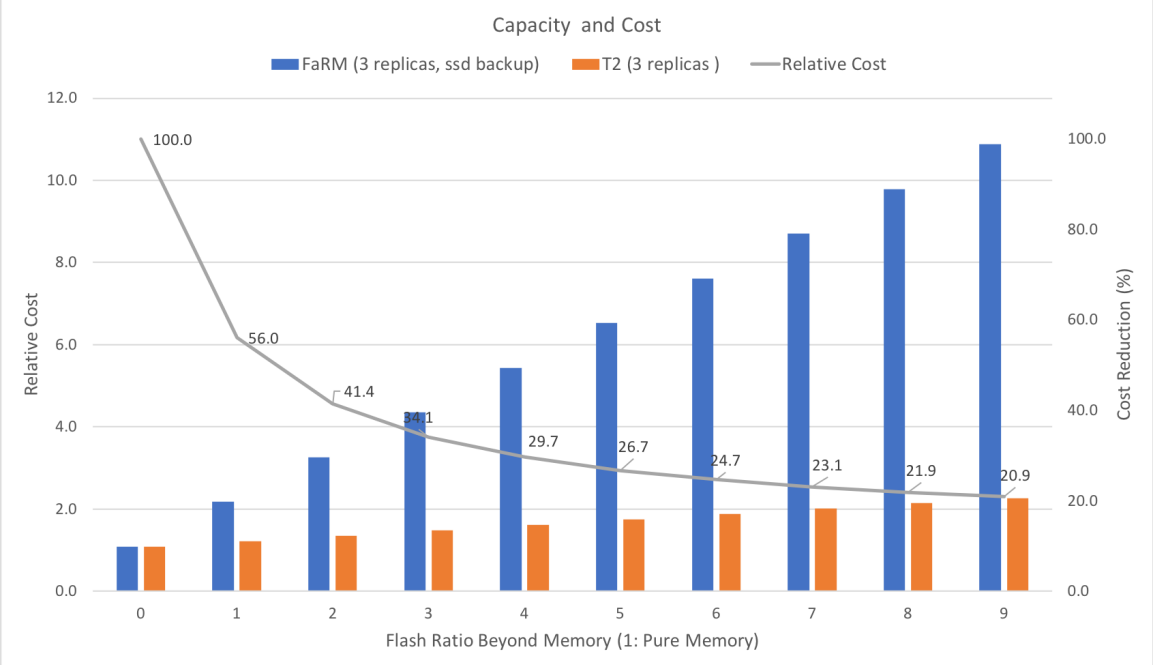


Figure 3.3: Relative Cost for Serving Beyond The Memory Capacity. To see the overall cost effectiveness, we use the terms *relative data capacity* and *relative cost*. For example, in a typical FaRM configuration, each server may offer 128 GB DRAM as usable data capacity to users (primary) and keep 256 GB SSDs for availability (backup). Having N servers with the same SKU, the total usable data capacity is N x 128 GB. We regard the relative data capacity of this configuration as one. In this graph, the x axis represents the relative data capacity. Similarly, we calculate the cost for DRAM and SSDs in this configuration with the current market price (Table 3.2) and regard it as the relative cost, one. The blue bar represents the relative cost of FaRM with one memory replica (primary) and two SSD replicas (backup), and the orange bar indicates the relative cost of the proposed system. However, this does not include the additional infrastructure overhead to hold the additional memory. The dollar per byte for SSD is significantly lower; the additional cost of T2 to provide more data capacity is much lower than that of FaRM.

model, which complicates the application logic and is prone to introducing subtle bugs.

To provide common programming APIs for both memory and flash storage tiers, we should resolve the interface disparities between the memory and flash storage. For a flash-tier subsystem to be compatible with the transaction memory-tier subsystem, two major disparities are the difference in data access interfaces and the lack of transaction support in commodity SSDs.

The recent non-volatile memory (NVM) technology, or storage class memory, dilutes the barrier between the memory and storage devices [56, 57, 58]. This can simplify the design of the unified interfaces through consistent filesystem-level interfaces for NVM, such as PMFS [59] or BPFS [60]. However, NVM adoption in industry is still very limited and not widely available.

In addition, recent SSD research shows the viability of transactional SSDs [52]. The approach is to extend the SATA interface for write and read commands and add commit and abort commands to change the transaction status. Although this approach can provide both simple programming interface and performance improvement for the transaction, they are not yet available as commodity SSDs that can be deployed in datacenters.

Consequently, resolving these disparities is one of the main topics in this dissertation. Specifically, FaRM provides memory-like programming interfaces for the globally shared address space with transaction semantics (Figure 2.3), and the interfaces are simple to use for programmers. Therefore, our goal is to keep such convenient interfaces as they are.

3.3.1 Storage Access Methods

The first disparity comes from the access method to storage devices. Operating systems provide two types of system calls to access flash disks: device I/O control (ioctl) or file system calls. Ioctl allows users to send block device commands through SATA [61] or NVMe protocols [62], which communicates directly with the storage devices. This may give more flexibility to applications to control devices; the direct use of the block-level in-

Table 3.3: Comparison of File Access and Memory-Mapped File Access.

	File Access	**Memory-Mapped File Access**
Access	read()/write()	memcpy, memory indirection
Overhead	system calls additional data copy	no system calls once data is fetched page fault
Buffer	extra buffer and copy required	no extra copy
Continuation	Easy to control asynchronous I/O	No asynchronous control support

terfaces can be very complicated because system designers should manage consistency and security of *data-at-rest*, which are critical in building large distributed systems. Therefore, system designers in large distributed systems mostly use file abstraction to access storage devices, and we also take the file abstraction as the low-level storage primitives.

Operating systems provide two different mechanisms to access and manipulate files in storage devices: file access and memory-mapped file access (Table 3.3).

The file access approach is a widely used way to implement storage systems [24, 40]. Applications simply open a file via system calls and use the returned handle or file descriptor to read and write data at a specific offset in the file. System designers can also explicitly control asynchronous input and output (I/O) operations. However, these file access operations incur expensive system call overhead each time as well as an additional data copy overhead between kernel-space memory and user-space memory.

In contrast, the memory-mapped file input and output approach maps a file segment into the application's virtual address space, which is executed through *mmap* in POSIX-compliant systems or *CreateFileMapping* in Windows systems. Once a file segment is mapped into a virtual address space, there is no additional system call and copy is not required between kernel-space memory and user-space memory unless the data is swapped out. Therefore, this approach often provides a more efficient mechanism of file access when the memory is sufficient for the working set. Once the file segment is mapped into the application's virtual memory space, applications can access the data through normal memory access operations, such as *memcpy* and memory indirection.

However, this improvement is not guaranteed in the *data-beyond-memory-capacity* context for the following reasons. First, the DRAM capacity to serve the underlying storage capacity is relatively small, for example, seven times smaller in the above server configuration. It cannot load all the memory-mapped data that are randomly accessed, and heavy random access causes frequent page faults and the performance degradation [63]. Consequently, in datacenter applications the number of page faults can easily nullify the performance benefits.

In addition, this approach is not sufficient to achieve performance in concurrent and asynchronous I/O heavy contexts as it is in data-intensive environments. In multi-core environments, exploiting asynchronous I/O operations is an essential technique to increase performance by running multiple tasks concurrently. However, the memory-mapped I/O mechanism only provides limited ways to control asynchronous continuation.

Last but not least, this approach yields its consistency control between a memory segment and its mapped file to the virtual memory manager. Because I/O errors resulting from storage device failures are common in datacenters, it is critical to control data consistency by storage systems themselves to guarantee correct transaction. These disadvantages of limited asynchronous support and controls outweigh the advantage of having a simple access mechanism for highly data-intensive distributed storage systems. Therefore, this dissertation uses the file access approach to build a memory-compatible data model.

3.3.2 Transaction Support

The next disparity comes from the lack of transaction primitives in commodity storage devices. Specifically, atomic operation support is the fundamental requirement to implement a transaction protocol. For example, an efficient transaction in FaRM is realized by exploiting atomic operations, such as *compare-and-swap*.

The transaction-replication protocol, or commit protocol, in FaRM requires two types of transaction primitives during the commit phase, which are the lock operation to change

the status of an object and the commit operation to update the data of an object as well as the status of an object.

3.4 File I/O Performance Characteristics

In a storage system requiring a file access during a normal operation, the overall performance depends on the file I/O. Although the performance of SSDs has improved over time, the latency of SSDs is still three or four orders of magnitude worse than that of DRAM (Table 3.4) and their throughput is often limited by the host interfaces, such as SATA or NVMe. While most systems utilize a memory buffer or cache to hide I/O latency and amortize the performance overhead [24], their actual performance can be varied by the underlying use of file systems. In this section, we explore the performance characteristics of Windows file system APIs [64] because they are the building blocks to implement a buffer or cache in the flash-tier subsystem.

Table 3.4: Latency comparison.

	Read Latency	**Write Latency**	**Erase Latency**
HDD	5 ms	5 ms	N/A
SSD	25 us	500 us	2 ms
DRAM	50 ns	50 ns	N/A

A reproduced version of three types of "storage" latency [65].

3.4.1 File I/O Measurement

Windows systems provide file access APIs, and application developers can set parameters to meet their performance requirements. The performance can be varied based on how files are accessed (synchronously or asynchronously), whether or not they are buffered (buffered or direct I/O), and the number of files that are accessed.

In this experiment, we measure the performance of different file I/O mechanisms available on the Windows system. We built a benchmark, *FileIO Bench*, to explore the perfor-

mance characteristics of different file I/O configurations available on the Windows system.

First, FileIO Bench creates a number of fixed-size files and generates a sequential or random access workload based on the two parameters: number of files and file sizes. The workloads consist of a list of *(file index, offset)* tuples. Then, FileIO Bench issues an I/O request for each tuple. The file index represents the index of a target file, and offset is an offset in the file. For example, the request of (4, 0x120000) will access the offset 0x120000 in the fourth file. The offset is the multiple of sector size (512 or 4096 KB) because Windows systems require alignment when we access files without filesystem caching ([64]). Also, the benchmark pre-allocates an aligned memory for simplicity.

The configuration used for experimentation has 16 GB of DRAM and Intel(R) Xeon(R) CPU E5-2620 v2 CPU and runs Windows 10 Enterprise, which uses the NTFS file system. We enabled hyper-threading (total 12 hardware threads). It has two SSDs (Samsung SSD 850 and Samsung SSD 860) that are connected with SATA 3.0 interfaces; therefore, the maximum bandwidth is 6.0 Gbps, which is 600 MB/s. The benchmark creates and accesses files in the SSD 860 model SSD. It issues 10,000 I/Os and we measure the total elapsed time and calculate the average bandwidth.

Table 3.5: FileIO Bench Parameters.

File Size	Number of Files	Threads	Buffer Size	File Cache	Sync/Async
256, 512 MB	1 - 2048	1 - 12	128KB - 4MB	On/Off	Sync/Async

3.4.2 Filesystem Cache and Number of Files

When an application issues a read or write I/O, an operating system may buffer or cache the data to improve performance. In Windows, if a file is opened via *CreateFile()* with the *FILE_FLAG_NO_BUFFERING* flag, Windows turns off the system caching and all the data of target files is read from or written to the storage device directly (Direct I/O). Otherwise, the Windows system keeps, or caches, data that is read from the storage device into the file system's internal *cache* and uses the *buffered* data in the cache instead of

accessing the file again (Buffered I/O).

Figure 3.4 shows the relation between the performance and the number of files. In a simple scenario, where an application accesses a small number of files (x-axis, toward left), the performance improvement is dramatic because there are many chances for the data to read from the file system cache within a small working set. In our experiment, the total bandwidth goes up to 6,000 MB/s. However, in data-intensive application scenarios workloads are generally spread over multiple files, and the performance improvement by the file system cache rapidly drops. In our benchmark, when the number of 512MB files is more than 200, which is about 100 GB, its performance improvement is negligible.

In a data-intensive application, a storage system needs to hold a few hundred GBs to a few TBs of data, and for ease of management and performance improvement, these files are stored in multiple *log* files. Therefore, the benefits of file system cache notably decrease. Moreover, similar to *Memory-Mapped Files*, applications have limited control of the cached data, which will impact the data correctness during machine failures.

3.4.3 Available Physical Memory

The performance improvement of the buffered I/O is related to the physically available memory in the system. FileIO Bench samples the memory status through *PERFORMANCE_INFORMATION* [66], which provides Windows' internal memory statistics. In particular, we measure PhysicalTotal, PhysicalAvailable, and SystemCache. PhysicalTotal shows the physical memory in pages, and PhysicalAvailable shows the physical memory that can be used immediately. SystemCache shows the memory that is used for the system cache.

Figure 3.5 shows the memory status when FileIO Bench is configured to access 1024 512-MB files through 4 MB buffers. In the buffered I/O case, as soon as FileIO Bench starts, SystemCache keeps growing to hold the data and as a result its available physical memory decreases. In contrast, Direct I/O has little impact on physically available memory and system cache. However, the file system cache is *standby* memory and Windows will

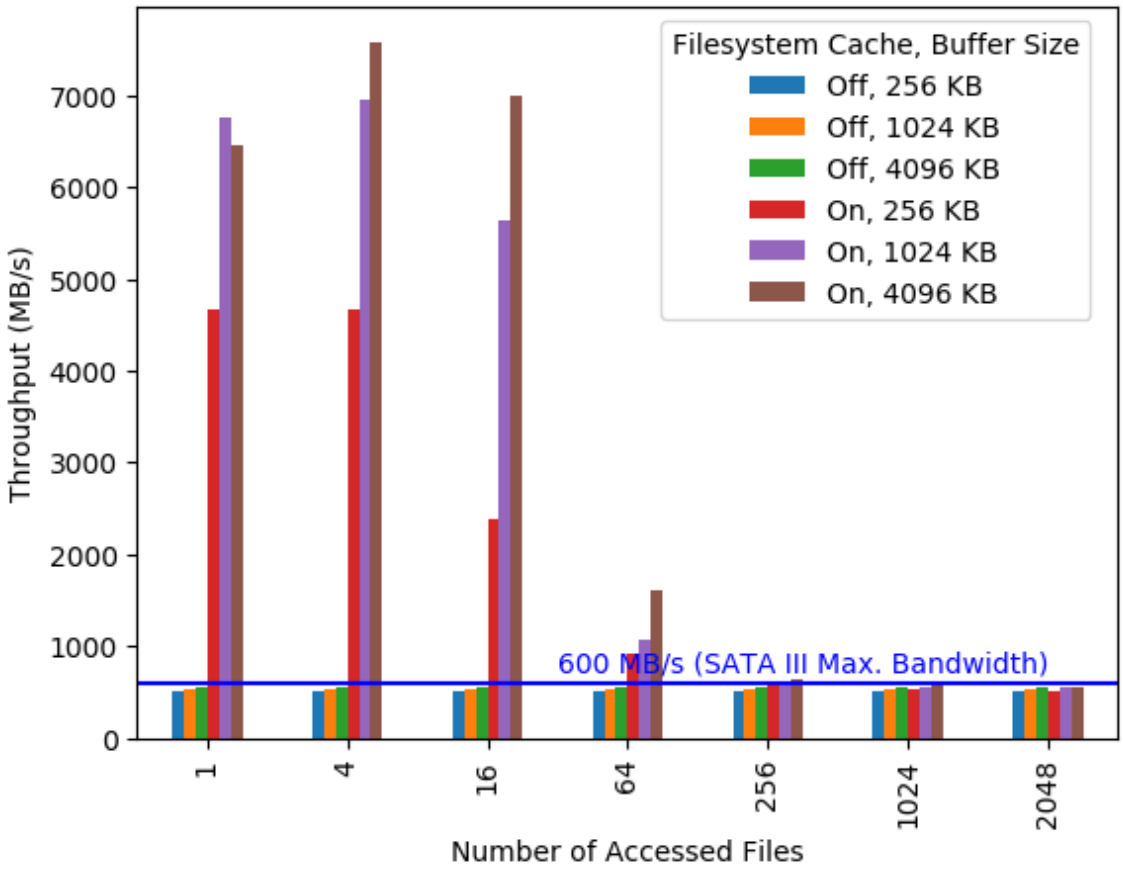


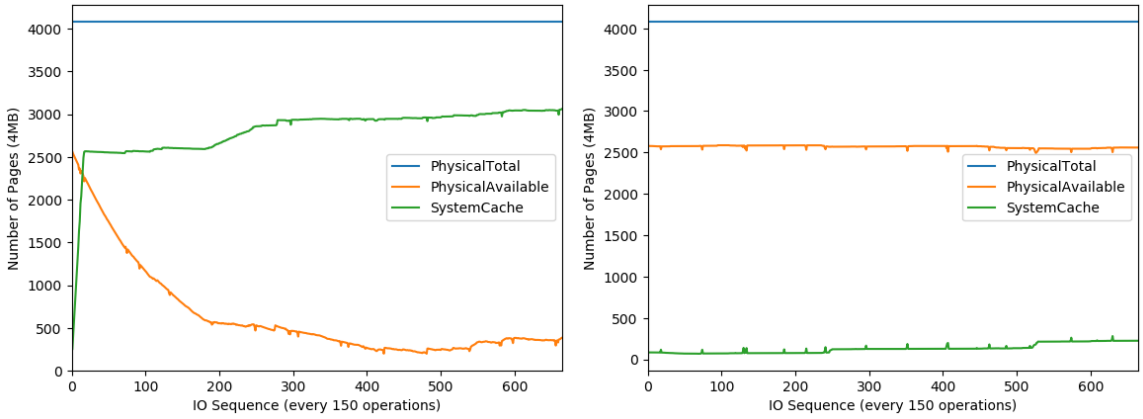
Figure 3.4: Read Throughput vs. Number of Accessed Files. When the number of accessed files is small, the file system cache keeps the previously accessed files in memory and the overall performance improves significantly. However, as the number of files accessed increases, its performance improvement dramatically decreases. We run 12 threads and each thread uses four queued buffers for concurrency.

use it when memory allocation is requested. However, this cannot be used immediately and needs additional operations to be used by the requester.

As a result, in data-intensive workloads Direct I/O offers a controlled way to use DRAM and allows other systems to use memory more efficiently.

3.5 Summary

In this chapter, we addressed two primary goals of the proposed tiered system: to make it cost effective *and* to preserve simple programming model provided by the existing in-memory system. We argued that providing memory-compatible access methods from the



(a) Buffered I/O. **(b) Direct I/O.**

Figure 3.5: Available Physical Memory and Filesystem Cache Size. FileIO Bench accesses 1024 256-MB size of files with 4 MB buffers. In this configuration, the total data footprint can be up to 256 GBs and the performance for both is similar (Figure 3.4). Because the file system cache tries to keep data *just in case* in memory, its physically available memory dramatically decreases. In contrast, Direct I/O has little impact on the physically available memory

commodity storage is a key factor to meet these goals.

We reviewed the cost and capacity aspect of tiered storage system configuration and then examined the interface disparities between the memory and flash storage that need to be overcome to support the memory-compatible interfaces. Among different storage access methods, we chose to use file abstraction rather than memory-mapped I/O mechanisms for better control of asynchronous operations and durability, which are critical to building a multi-concurrent datacenter platform.

We then examined the performance characteristics of file I/Os in Windows systems. In particular, we compared the performance between buffered I/O and direct I/O mechanisms, varying the size of the total data footprint by increasing the number of files. While buffered I/O can improve overall performance when most data can be cached in, its improvement is negligible in data-intensive workloads.

CHAPTER 4
TIERED TRANSACTION STORAGE SYSTEM

In the previous chapter, we considered storage subsystem primitives for a tiered transaction storage system, which offers memory-compatible programming interfaces. This chapter presents an architecture of a proposed *T*iered *T*ransaction storage system, T2, to achieve cost effectiveness *and* ease of programming. We go on to explore the various options of an object model that the flash-tier subsystem offers and discuss their advantages and disadvantages. T2 provides a *visible* and *static* object model to users, identifies an object with an *id-based* address, and realizes it through a virtualized *region* instance.

4.1 Architecture

T2's architecture extends FaRM's architecture to utilize the memory-compatible primitives and transaction protocol. These primitives enable the proposed system to provide common programming APIs to users and allow the flash-tier subsystem to exploit FaRM's efficient transaction and replication protocols (Figure 4.1).

As an extension of FaRM, T2 shares the common components of FaRM, such as memory allocator, local and remote data access manager, and transaction/replication manager. A separate coordination service, or *zookeeper*, manages the globally shared configuration shared by each server.

To accommodate the flash tier as a compatible tier, T2 then extends the logic of memory allocation to support transparent access to both the memory tier and flash tier to users. An application that is running on any machine in a cluster can access any object stored in memory and in flash tiers spanning the cluster.

A large portion of DRAM in a machine is pre-allocated for the memory tier through *Pyco* driver [9], and some portion of the remaining DRAM is allocated for the flash-tier

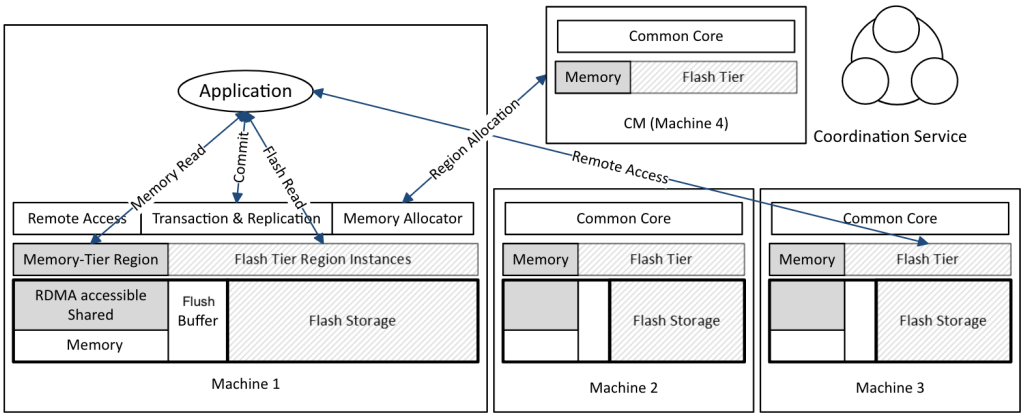


Figure 4.1: The Architecture of T2. The architecture of a tiered transaction distributed storage system, or T2. An application is running on a machine and can allocate and access objects that span multiple machines. Every machine is equivalent except that the CM machine orchestrates the cluster-wide information, such as region allocation and server configuration changes. A part of the physical memory is pre-allocated for the shared memory tier, which is registered for RDMA access. Each flash-tier instance also uses a portion of DRAM for its internal flush buffers and caches.

subsystem to improve performance. In this dissertation, the flash-tier subsystem design also relies on FaRM's non-volatile DRAM backed by the distributed UPS [9], which simplifies the recovery logic for durability.

4.2 Programming Model

While it is preferable to keep the same programming model of the baseline system as much as possible when extending an existing system, it is often necessary to allow some changes in the programming model to meet the holistic system goals. T2's programming model introduces a change to deal with the heterogeneity of two tiers, or the object model. This section discusses the modified programming model, and the next section discusses in detail the change and multiplicity of different object models.

To enable the transparent memory- and flash-tier access, T2's flash-tier subsystem supports the programming model similar to FaRM (section 2.4.1). In FaRM, an object is allocated from a global memory space spanning machines, and applications running on any machine can access local or remote objects designated by an address transactionally via the programming interfaces (Figure 2.3).

```
Tx* txCreate();
void txAlloc(Tx *tx, int size, Addr addr, StorageAttr attr, Cont *c);
void txFree(Tx *tx, Addr addr, Cont *c);
void txRead(Tx *t, Addr a, int size, Cont *c);
void txWrite(Tx *t, ObjBuf *old, ObjBuf *new);
void txCommit(Tx *t, Cont *c);
```

Figure 4.2: T2's Modified Version of FaRM API. T2 provides an almost identical APIs that FaRM [9] provides for simple memory management and transaction support. *txCreate* creates a transaction context for the following operations. *txAlloc* allocates an object from the designated tier's shared global space. T2 extends the original allocation interface by including a storage attribute, which designates the residence of an object. *txFree* frees an object. *txRead* reads object data from the globally shared space. *txWrite* updates an object in its local heap first, and then the update is applied to the shared space only when *txCommit* completes successfully.

While T2 supports this programming model as a baseline, it addresses the heterogeneity of the media explicitly. In T2, memory-tier and flash-tier address spaces are exclusive. A user should directly specify which tier to use at the allocation time via a storage attribute. The residence of an object is then statically determined, and the allocated object lives in the same tier for its entire lifetime. It is possible to provide the same interface without having a separate storage attribute. While this approach can be more transparent to application developers, it has performance implications, which will be discussed in the following sections. Accordingly, T2 slightly changes the interface of allocation with a new parameter: storage attribute.

In the memory tier, an object occupies an address space taking at least the size of the object or more, although it is not required in the programming model. However, this virtual space of an object imposes inefficiency to manage an arbitrary object in the flash-tier address space. Therefore, in T2, when an object is allocated, it gets a globally unique identifier, or object id, that is used by the flash tier. In this dissertation this object id is referred to as the flash-tier's *address*, but it is not related to any physical location or logical size of an object. All the remote and local accesses to objects in either tier are transparent to the application.

Once an application allocates objects from either or both tiers via a storage attribute, it

can execute arbitrary logic against the object, such as write, read, and free, regardless of their different characteristics within the transaction. Then, when the application commits the transaction on any arbitrary machine, the thread that receives the transaction request on the hosting machinebecomes a coordinator for the transaction.

In the following section, we explore the design space of an object model to support the consistent programming model from the heterogeneous underlying media.

4.3 Object Model

An object model addresses how the system represents an object to users and to internal subsystems, and there are multiple options (Table 4.1) regarding how to represent an object to users and how to realize the object in the system. This section focuses on the following four options: externally, 1) the system may expose the difference of object types to users (*visibility*), and 2) it may support the change of the object's residency (*dynamics*) over time. Internally, 3) it may represent an object in a contiguous address space or discrete address space (*address*), and 4) it may manage the objects in different granularity (*granularity*).

The following subsections consider the pros and cons of each choice. Each subsection first addresses general research implications for each option and then discusses the related system constraints or goals of the dissertation and concludes with a decision.

Table 4.1: Object Model Design Space.

Visibility	**Dynamics**	**Address**	**Granularity**
Visible*	Dynamic	Object ID*	Region*
Transparent	Static*	Offset	Object

The table shows the different options for an object model for the flash tier. The asterisks (*) indicate the choice for T2. Externally, a user sees the differences of an object-hosting tier (*visible*). Once an object is allocated from a designated tier, the object stays there during its lifetime (*static*). Internally, an object is managed by the unit of a region (*granularity*) rather than by a small individual object, and its address is interpreted as an object id (*address*).

4.3.1 Visibility - Visible vs. Transparent

The first design space of an object model is about whether to expose the differences of each object type to users and we have two options: *transparent* and *visible*.

- *Transparent*: a user has no knowledge of the location of objects, and the medium of the objects is chosen by the system and may change over time.
- *Visible*: a user has explicit control of deciding the location of objects, such as which objects can be stored in the memory tier or flash tier.

The advantage of the *transparent* option is that it provides the same programming interfaces to users and the system manages all the allocation based on some criterion; application code does not need to change to use an additional flash-tier subsystem.

Nevertheless, this option gives users little control of object residence, and applications can experience unexpected performance degradation, depending on the workloads and data access patterns. In addition, it increases the complexity of the system implementation because the system needs to understand the application's data access pattern and its life cycle in detail. The system's failure to choose the *right* place for an object negatively affects the performance and/or wastes the expensive space of memory. It can implement a specific logic for the residence of objects that are applicable to common workloads; for example, for a workload where the LRU caching policy fits well, this *transparent* approach is convenient to application developers. Unfortunately, this approach is not optimal for a common datacenter workload because the datacenter workload usually involves massive data requests with different access frequencies [31].

In contrast, the advantage of the *visible* view is that it is widely adopted as common practice exercised by datacenter applications. For example, in building social graphs [25, 2], the type of objects, their usage patterns, and performance requirements are already known to developers in advance or over time. For example, such social graphs often consist of *meta data* that contains *meta* information of a person, such as name and relationship, and

regular data, such as multimedia objects. The meta data are generally small and accessed frequently and require low latency all the time. In contrast, the regular objects, such as multimedia data or documents, are relatively large, are accessed less frequently, and can tolerate more latency. The disadvantage of this option is that existing applications need to be modified to get the benefits of the tiered transactional storage system.

T2 took the *visible* approach for several practical reasons. From the datacenter application context, every application has its own workload and the guaranteed performance is often preferred to unreliable performance [67]. The visible approach allows developers to estimate their performance characteristics in advance for a target application and prevent any surprises of unexpected data relocation by the system. Application developers can optimize their object management logic as they do in their current practice. Finally, this approach also simplifies the allocation and access logic of the flash-tier subsystem.

4.3.2 Medium Change - Static vs. Dynamic

The second object model option is whether to support the change of the object's residency over time, and we have the following options: *static* and *dynamic*.

- *Static*: the resident medium of an object is determined at the allocation time and does not change over time.
- *Dynamic*: an object may be migrated between the memory and flash tier over time, possibly based on access pattern.

The *static* option supports the expected performance guarantee offered by the *visible* option; however, it does not solve the dynamics of workload and access patterns. As we can see, datacenter workloads and access frequency change over time [15]. For example, the information about a recently released movie will be requested frequently (*hot*), but the number of requests will decrease (*cold*) over time.

The *dynamic* option allows a storage system to maximize utilization of its memory resources and provides a more cost-effective method over time. An object can be allocated in one of the tiers implicitly (*transparent*) or explicitly (*visible*). A storage system can then migrate infrequently accessed objects, or *cold* data, into the flash tier while moving frequently accessed objects, or *hot* data, into the memory tier.

However, similar to the *visibility* option, the *dynamic* option needs more sophisticated logic to prevent *surprises*. Although with reliable migration logic the *dynamic* option provides better cost effectiveness over time, we leave this option for future work and limit our work to the *static* option. Instead, with T2, users can implement their data-migration logic and explicitly handle it inside the application.

4.3.3 Address - Offset-based vs. Id-based

The following two design options are related to how they are implemented internally in the subsystems.

The third object model option is whether or not to support an offset-based address. The following are the two available options: *offset-based* and *id-based*.

- *Offset-based*: an address is meant to be an offset in a region in the globally shared memory space.
- *Id-based*: an address is an opaque identification number, *id*, pointing to a specific object in a region. It is not related to any logical offset in the shared address space.

While the memory tier internally exploits the virtual memory address, there is no intrinsic mechanism in flash disks. The flash-tier subsystem should therefore provide an *address* for memory compatibility.

The advantage of the *offset-based* option is that it is very intuitive to manage objects because the internal object model is the same as that of the memory tier. It also eases op-

timization of some operations, such as array-based operations against contiguous objects. However, this option complicates the allocation mechanism of the flash-tier system. For example, it needs to manage address space through special allocation memory techniques, such as slab alloc, to allocate different sizes of objects efficiently and to prevent fragmentation. In contrast, the *id-based* option shows flexibility to cope with different sizes of objects, reducing worry about the fragmentation in the address space. Objects can be allocated and freed independently regardless of their size and *address*.

We used the *id-based* option because it offers more flexibility to manage objects inside the flash tier. With this approach, we can also make the flash-tier subsystem more flexible so it can support other in-memory storage systems without much change. Although T2 uses a hybrid address for the internal object identification (Figure 4.3), the address is externally opaque to applications.

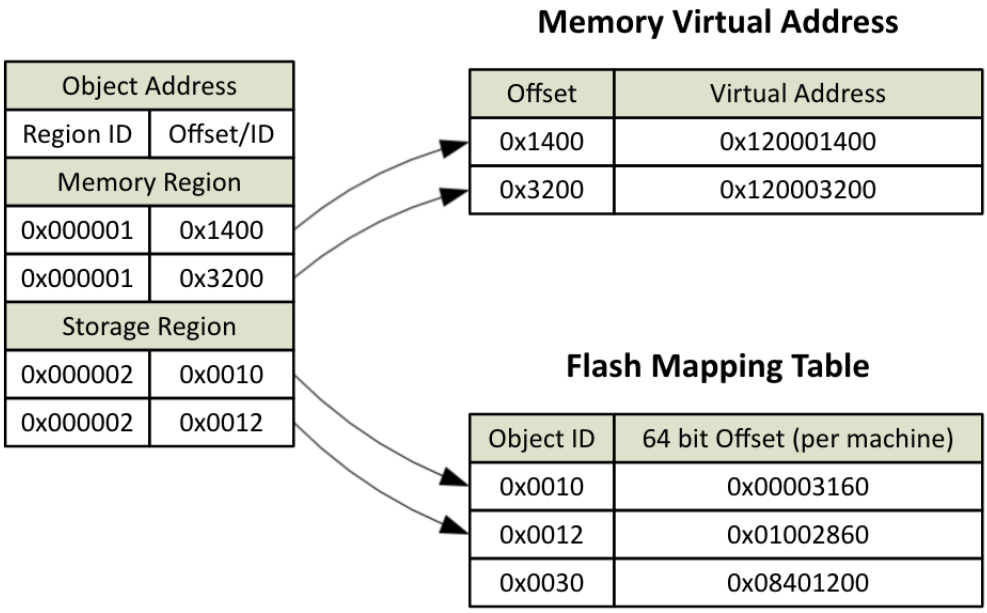


Figure 4.3: Hybrid Address Interpretation. T2 uses a hybrid approach to interpret an address. An address for the memory tier is similar to virtual address interpretation; however, an address for the flash tier is an opaque handle, or an object id.

4.3.4 Management Granularity - Region vs. Object Level

The last object model option addresses the management of objects in T2. The following are two available options: *object level* and *region level*.

- *Object level*: an object is the unit of residence and management on the memory tier or flash tier. With this approach, any object can be easily migrated into a different medium.

- *Region level*: a region is the unit of residence and management on the memory or flash tiers. With this approach, the existing management and recovery mechanisms are not affected for the memory tier.

The fine granularity, or *object level* management option gives more flexibility to both users and subsystems. For instance, an object can be easily migrated between the two different tiers without changing their exposed *address*. Then, an object can be moved from the memory tier to the flash tier implicitly or explicitly, which improves overall cost effectiveness. However, it increases the complexity of the subsystem and adds management overhead.

In contrast, the advantage of the coarse grain, or *region level*, is the simplicity of the implementation. With this coarse granularity, the modification for the internal mechanism for allocation and recovery can be minimized. The memory tier manages its address space by the unit of a *region*, which allows the system to overcome the limited DRAM capacity in NIC [9]. This allows T2 to manage the flash-tier subsystem independently from the memory tier. The disadvantage of this option is that whenever an object is migrated from the memory tier to the flash tier or vice versa, it would require an explicit address change by users, who should implement the migration logic with the provided transaction primitives.

As discussed in the previous *visibility* subsection, our scope focuses on providing an explicit way to deal with performance characteristics and preventing unexpected data relocation by the system; we therefore limit our discussion to the *region-level* option.

Given the four types of object model options, we took the *visible*, *static*, *id-based*, and *region-level* approaches. This set of options minimizes system complexity and allows users to more explicitly trade off performance and cost.

4.4 Region Abstraction

To provide the object model from the flash storage device, T2 manages the objects in the unit of *region*. *Region* is a maximum unit of memory and recovery management that is configured to overcome the limited size of the memory in NIC. T2 abstracts the *region* to be applicable to the flash tier such that the flash-tier *region* can also be used in the memory and recovery management. While the region of the memory tier is a partitioned address space that is mapped to the virtual address space in a local machine, the region of the flash tier is a partition of logical storage space. It also provides data management and transaction primitives to the internal system.

4.4.1 Implication of Region Abstraction

The region abstraction provides the memory-compatible interfaces, and it offers several advantages over existing hybrid approaches (section 2.8), where application developers manage both in-memory and flash-based storage systems manually to overcome the disparities between two heterogeneous systems.

The benefits of region abstraction are as follows: first, region abstraction allows developers to use the same APIs (Figure 4.4) for both memory and flash tiers. Developers specify which tier to use during the allocation phase. Once an object is allocated and its address is given, all the data access mechanisms are the same for both tiers; there is no additional change in the application logic. All the objects in memory or flash tier can be accessed within a single transaction, and the transaction guarantee of FaRM, ACID transaction with strict serializability remains the same.

Second, the region abstraction internally simplifies the design and implementation of

the flash-tier subsystem. The existing transaction and replication commit protocol involves several atomic operations to validate the version of objects and to update the content of objects that are being committed. The region abstraction allows the flash-tier subsystem independently to provide those atomic operations and reuse the transaction and replication protocols of FaRM.

Each logical region of the flash-tier subsystem has two types of regions to support the replication protocol: primary and backup. During normal operation, applications can read an object that is in the primary region. The backup regions are updated only during the commit phase in a normal case and are accessed during the recovery phase. The backup

```
1   struct Meeting {
2     obj_addr date;
3     obj_addr attendees;
4     obj_addr documents;
5     obj_addr prev;
6   };
7
8   Meeting meeting;
9   Transaction tx = new Transaction();
10  meeting.date = alloc(tx, data, memory);
11  meeting.attendees = alloc(tx, data, memory);
12  meeting.documents = alloc(tx, data, flash);
13
14  // Read the previous meeting data
15  meeting.prev = read(tx, addr);
16
17  // Update the meeting contents
18  write(tx, meeting.date, ``12/11/2017, 3PM-5PM''); // memory tier
19  write(tx, meeting.documents, large-size-meeting-notes) // flash tier
20
21  tx.commit();
```

Figure 4.4: Example of Application Code in T2. With the region abstraction, T2 extends the simple programming APIs of FaRM to manage both memory and flash residing objects. This example shows a simplified version of T2 APIs; instead of having a continuation parameter for asynchronous operation, we used synchronous operations. An application specifies the target medium during allocation, lines 10-12. It then retrieves the previous meeting data, line 15, and updates the current attendees and documents, lines 18-19. Except for the first allocation statements, all other statements use the same interfaces used for the in-memory version and users can perform the operations in a single transaction context.

regions are located in different servers to provide high availability and durability in case of machine crashes.

Figure 4.5 shows the simplified version of the implementation of the read access for T2. Each server keeps a region table, and T2 retrieves a handle for a target region. When the region belongs to the flash tier, the handle represents an address of the corresponding region instance. The thread then requests a read operation to the instance, and the flash-tier region instance processes the request asynchronously.

4.4.2 Realization of Region Abstraction

A flash-tier *region instance* is a running thread that realizes the region abstraction for the flash tier and processes data management and transaction operations (Figure 4.6).

The flash-tier region instance can be created in advance when T2 starts or on demand when there is no available space in the flash-tier regions. When a new region needs to be created, a transaction coordinator requests a new region with the target *storage attribute* to

```
1   // For the memory tier
2   void txRead(Tx *t, Addr a, int size, Cont *c) {
3     region_id = get_region_id(a);
4     offset = get_offset(a);
5     dst_buffer = get_destination_buffer(c);
6     base_addr = region_table.get_handle(region_id);
7     if (is_memory_tier(region_id)) {
8       src_addr =  base_addr + offset;
9       memcpy(dst_buffer, src_addr, size);
10      // Process the continuation
11    }
12    else {
13      flash_region_instance = (FlashRegion *)base_addr;
14      flash_region_instance->read(dst_buffer, offset, size, c);
15    }
16  }
```

Figure 4.5: Simplified Implementation of Read in the Subsystem. The code shows the simplified version of reading an object. A region table is shared in the system and returns the base address, which is the base virtual address for the memory tier or region instance address for the flash tier. Unlike the memory read operations, reading from the flash tier is asynchronous; therefore, it accepts the continuation c as a callback parameter.

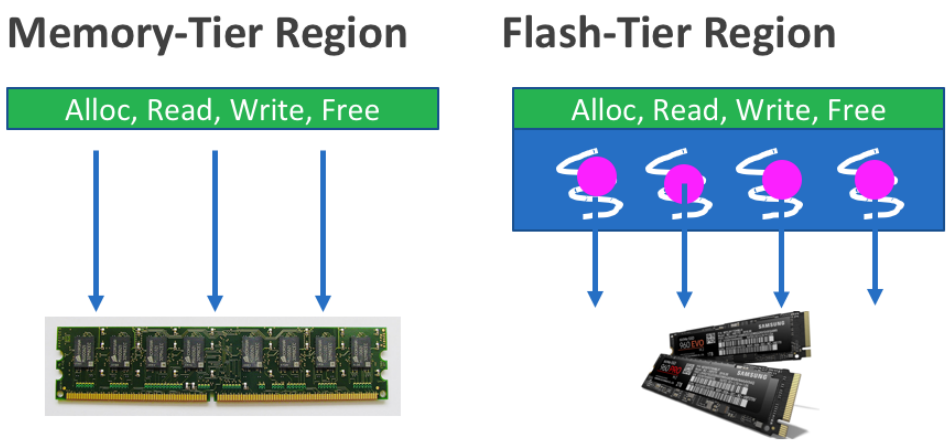


Figure 4.6: Flash-Tier Region Instance. The memory-tier region (left) uses conventional memory access mechanisms; data are stored in a static location in virtual memory space and accessed through a pointer, or an address. To provide similar interfaces from the flash tier, T2 creates a *region instance* for each region (right). *Region instance* is a virtualized region running on a thread and stores data in flash storage. It converts the internal memory accesses to underlying memory operations on DRAM buffers or I/O operations on flash disks.

the configuration manager (CM). The CM then assigns a new region id from the available servers and sends it to the corresponding primary and backup servers. The new region information is shared with all the servers in the cluster. Each server keeps a cache of the region-mapping information for the newly allocated region. The servers that have the newly assigned primary and backup regions create new flash-tier region instances, and the region mapping table in the servers maps each region id to the local region instance's address, or handle.

To handle multiple concurrent allocation requests, T2 queues the allocation requests. The allocation process is asynchronous because it needs to communicate with CM through the network. Once a new flash-tier region is allocated, T2 dequeues the requests and assigns an address for each request.

Once a new flash-tier region is allocated with its region id, the flash-tier subsystem can assign an object from the flash region instance. The region instance generates a new object id for each request and combines the region id and the object id to return as an *address* to a user.

Table 4.2: Region Mapping Table and Flash-Tier Region Instance.

Region ID	Tier	Local/Remote	Handle	Description
1	Memoy	Local	0x18000000000	Virtual Address
9	Memory	Remote	0x8000170000	RDMA reference
12	Flash	Local	0x0040214310	Region Instance
20	Flash	Remote	-	-

In the memory tier, a region-mapping table keeps mapping information between a region and the base address for a local region and RDMA reference for a remote region. In contrast, in the flash tier, the address of a flash-tier region instance is kept for a local region only. For a remote region instance, any data access operation is requested through a regular RPC call, and the region table does not need to keep that information.

4.4.3 Region Address Mapping

When an application requests an access to an object, the flash-tier subsystem retrieves the corresponding region instance's handle from the region-mapping table and forwards the offset to the region instance for the requests. For the remote region, the flash-tier subsystem cannot access the remote region instance directly; it sends an RPC call to the flash-tier subsystem to process the request.

4.5 Summary

In this chapter, we described the architecture of T2 to achieve cost effectiveness *and* ease of programming. We then explored the design space of various object models, taking the *visible*, *static*, *id-based*, and *region-level* options for the object model. We extended the concept of region to abstract the discrepancy away and to provide consistent semantics with minimal code changes. Region instance is the realization of the flash-tier region that allows the system to support the same transaction and replication protocol with minimal changes.

CHAPTER 5

EFFICIENT FLASH-TIER SUBSYSTEM

This chapter discusses in detail the design and implementation of the flash-tier subsystem of T2. As discussed in the previous chapter, *region instance* is T2's approach to build a *memory-compatible* tiered transaction storage system with the flash storage. T2 exploits CAS operations to build transactional primitives, keeps the shared transaction status in the region instance's mapping table, and utilizes asynchronous I/O operations with flush buffers. These techniques allow the system to achieve cost effectiveness and ease of programming by minimizing the performance overhead incurred by the region instance and provides the same transaction guarantee.

The following section starts with a review of the current FaRM's commit protocol and considers the requirements for the flash tier and performance impact by the flash tier. The subsequent chapters describe the techniques used to meet the requirements and minimize the performance overhead.

5.1 Flash-Tier Subsystem for Commit Protocol

The flash-tier subsystem follows and implements the FaRM's commit protocol, which consists of two phases: execute and commit (Figure 5.1). A transaction starts with *txCreate* (Figure 2.3) and the first execute starts. A user then exercises arbitrary logic using *txAlloc*, *txFree*, *txRead*, and *txWrite* operations during the first execute phase. The second commit phase starts with *txCommit*. Internally, the commit phase includes four steps: *lock*, *validate*, *commit*, and *truncate*. As soon as the system completes the *commit* operation, or the data is stored in the non-volatile memory buffer backed by the distributed UPS, it can acknowledge (ACK) the transaction's completion to the user without losing durability.

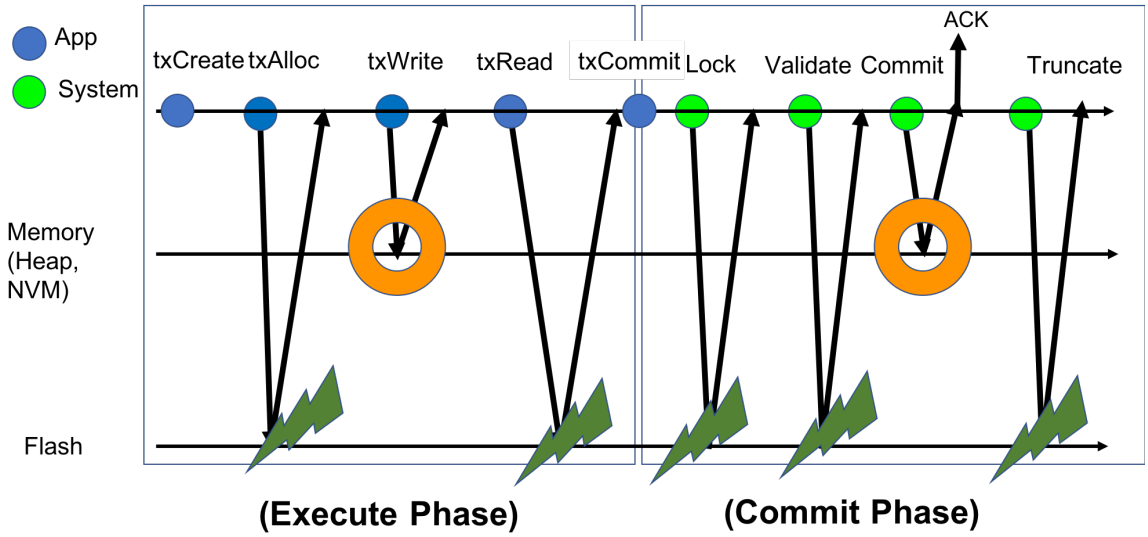


Figure 5.1: Commit Protocol For Flash Tier. This shows how the commit protocol interacts with the flash tier. The blue circle (App) represents the user-visible operations and the green circle (System) represents the internal system-visible transaction primitives. When a user calls *txCommit*, the commit phase starts. From the performance perspective, the *Write* and *Commit* primitives are already optimized because the system uses heap memory and non-volatile memory ring buffers; however, the other operations, such as read and truncate, need to be performed against the flash-tier subsystem. Therefore, T2 utilizes several techniques to optimize these paths.

5.1.1 Performance and Correctness Consideration

The flash storage directly affects the overall commit performance because it increases the overall latency and throughput resulting from the flash I/Os. Furthermore, it may change the correctness of the transaction because the commodity flash storage does not support the transactional primitives by itself. This section therefore addresses the performance and correctness implication in detail.

Figure 5.1 illustrates the flash-tier's commit protocol and shows the performance-related transaction primitives. Two operations, *txWrite* and *txCommit*, are performed the same way as the memory tier; *txWrite* is performed on heap-based local memory objects, and *txCommit* is performed on the non-volatile memory on remote machines. They are already efficient and not affected by the flash tier. However, *txCommit* to local memory is processed differently and needs to update the local memory directly. Consequently, flash storage's write performance has an impact on the commit performance.

On the other hand, all the other operations need to be implemented to minimize the flash access or to reduce the latency if possible. During the execute phase, *txAlloc* requires an efficient allocation and unique address retrieval from the flash tier.

txRead needs to access the flash storage directly and incurs high latency to applications in general, which is very common in datacenter workloads [31]. Although the overall performance in this phase is dominated by the read operations on the flash tier, it is also important to abort a transaction when an object is locked. For transaction correctness, the flash tier should guarantee the unique address allocation.

The commit phase is where the actual transaction and recovery protocol proceed. The performance of committing primary data to the flash-tier region directly affects the latency and throughput of the transaction because only after the completion of the commit does the transaction coordinator send its acknowledgement to users. While the memory tier's validation and lock operations can access the virtual memory space directly, the flash storage does not support such operations natively and the flash tier maintains a separate status for the lock and validation operations. As for the commit operations, the system keeps the committed messages containing the object data in the non-volatile ring buffers. The primary region's data is written to the flash-tier region when a server receives the commit primary message; however, the commit backup message is not applied until the messages in the ring buffers are truncated to the actual data location. For transaction correctness, the flash-tier subsystem must support the atomic lock and validation operations, and for the performance, it should process write operations efficiently.

Therefore, we can summarize the requirements for the flash-tier subsystem in the contexts of correctness and performance.

- **Correctness**: in order to keep the original transaction guarantee for committed read-write transactions, the flash-tier subsystem should provide efficient atomic primitives that are equivalent. In particular, *strict serializability* correctness relies on the relation between the version during the locking phase for written objects and the version

during the validation phase for read objects [18].

- **Performance**: in order to provide high performance, the flash-tier subsystem must reduce flash storage access and handle concurrent operations very efficiently. In particular, several transaction primitives can incur significant overhead; we focus on the optimization of transaction primitives and concurrency handling.

5.1.2 Supporting Primitives

To support the user-visible and system-visible operations, the flash-tier subsystem should implement the following primitives: *Allocate*, *Free*, *Read*, *Validate*, *Write*, and *Lock* (Table 5.1).

The system has two types of region instances for primary and backup regions, both of which have different roles. Users can access the data in primary regions and backup regions that are not exposed to users and used internally for durability and availability during a machine crash. Therefore, although both flash-tier region instances provide similar APIs, there are slight differences between the primary and backup region instances whether the system is in the normal mode or in the recovery mode. For example, the backup region instance does not need to allocate an object because the primary region instance already assigns the object id. As a result, the backup region instance does not have the allocate operation but needs to apply the object id during the commit phase. In addition, the backup region instance does not support the read operation during the normal mode, but it should support the read operation during the recovery mode.

A region instance is designed to separate the object's state and actual data. Some operations, such as read and write, may need immediate access to flash storage, and the other operations can be performed without accessing flash storage. This separation allows the flash-tier subsystem to support an efficient transaction.

Table 5.1: Flash-Tier Region Instance APIs.

Flash-Tier	Phase	Related operation	Region Type	State Access	I/O
Allocate	Execute	txAllocate	Primary	Update	No
Free	Execute	txFree	Primary, Backup	Update	No
Read	Execute	txRead	Primary, (Backup)	Read	Yes
Validate	Commit	Validate	Primary	Read	No
Write	Commit	Commit/Truncate	Primary, Backup	Update	Yes
Lock	Commit	Lock	Primary, (Backup)	Update	No

A flash-tier region instance provides the following APIs (Flash-Tier column) to support the transaction protocol. Each API operates on a different phase (Phase column) and is related to user-visible or system-visible operations (Related operation column). The region type column represents the type of region instance that supports an API, and the parentheses represent unavailability during the normal mode, but are utilized during the recovery mode. Some APIs access state (State access column) or data in flash storage and involve storage access(I/O). For example, read and write operations need data access and may introduce immediate I/O operations. The other operations can be processed without issuing I/O requests by managing object status separately.

5.1.3 Local and Remote Operations

The local and remote operations of the flash-tier subsystem are processed differently.

Flash-Tier Write Operation

When T2 processes *Commit* against a local region, it can directly access the corresponding flash-tier region instance and request the write operation because the transaction coordinator initiating the transaction is running in the same process. However, when an object resides in a remote server, it takes the same approach as the memory tier; the coordinator requests the update in the non-volatile logs or ring buffers first and then applies the update to the flash tier during the truncation phase. However, the ring buffer cannot be reused unless the data in the ring buffer is written to the flash tier completely. Consequently, the flash tier's efficient write operation is critical to both local and remote operations for efficient transactions.

Flash-Tier Read Operation

Traditionally, I/O latency could be reduced when there is a cache hit; however, the design principle of T2 excludes the possibility of using *auxiliary* memory cache (section 3.1). Therefore, the read I/O latency cannot be hidden from users. T2 focuses on the optimization of the read operation. As for the remote read operation, T2 cannot use the memory tier's approach using one-sided RDMA. Instead, it sends a request to read an object to a remote target region instance through a regular RPC mechanism, and the hosting region instance reads the object data and sends it back to its requester.

Figure 5.2 illustrates local and remote operations for the flash tier.

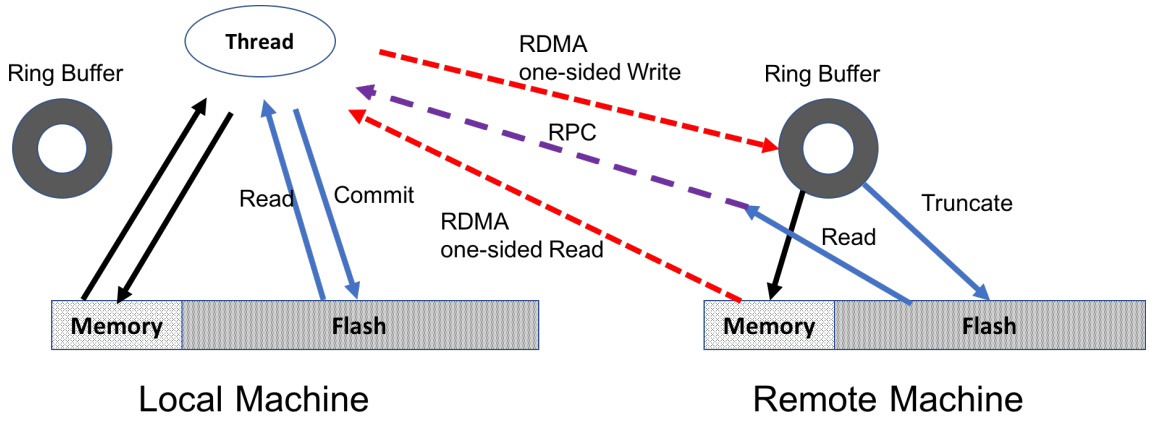


Figure 5.2: Local and Remote Operations For Flash Tier. The diagram shows the mechanisms of local and remote operations. The black line represents the memory tier's memory access, and the blue line represents the flash tier's flash access. The red dotted line represents one-sided RDMA operations and the violet dotted line represents a regular RPC operation. When a thread commits an object to a local object, T2 directly writes the object in the flash tier during *Commit*. However, when an object resides in a remote server, the update is written to the non-volatile ring buffer of the remote server first through a one-sided RDMA write operation; it is then applied to the flash tier during the truncation phase.

5.2 Transaction-Aware Mapping Table

The core data structure of a flash-tier region instance is a mapping table. To provide correct and efficient transaction primitives, it uses a *transaction-aware mapping table* to share the transaction state with the higher-level transaction primitives. It also utilizes flush buffers

with a log-structured scheme to reduce the I/O latency for write operations.

5.2.1 Mapping Between Address And Files

The flash-tier subsystem stores data objects into SSDs as a file and manages them in a log-structured manner [40, 22]. When a write request is received, the corresponding region instance writes the object in pre-allocated DRAM buffers, or *flush buffers*, and completes the request. Once the flush buffer is filled, the region instance *flushes* the buffer to flash storage devices asynchronously. The flush buffers convert random writes to sequential writes and maximizes the SSD's fast sequential performance characteristics. T2 minimizes the overhead of this process and achieves 96% of the maximum throughput (section 6.2). Each flush buffer has a header in the beginning that describes the status of the buffer, such as timestamp, buffer size, and garbage-collection related information.

As a log-structured system, the flash region instance's mapping table (Figure 5.3) first contains the basic *mapping* information between an object address and an offset in a *log* file. A region instance assigns an object id to an object, and the actual data must be stored at a certain location in a file. Each region may have multiple physical files, or logs, and the total size of the log file is multiples of the region size so that it can perform garbage collection efficiently. An object can reside in a flush buffer and/or in a log file, so in order to manage both DRAM buffer and flash file efficiently, the flash-tier region instance utilizes *virtual flash offset* [40]. Virtual flash offset is an offset in a virtual address space, and each region instance has its own address space. If an object is located in a per-region virtual flash address space, or logical address, its virtual flash offset shows whether it exists in a flush buffer or in the corresponding log file. The logical address of a log file starts from 0 offset, and the flush buffers are always in the currently maximum virtual address blocks (Figure 5.3). For example, in Figure 5.3, when the size of one flush buffer is 1MB, it can hold the address space of 0x100000. The flash-tier region already has multiple log files up to 0x8400000 exclusively, and the first flush buffer's address starts from 0x8400000.

With this mapping scheme, the flash-tier subsystem can support an arbitrary size of objects rather than support a fixed size of pages; however, for simplicity, we design a system such that the maximum size of an object is less than that of the flush buffer.

5.2.2 Mapping Table For Transaction

The above mapping table provides a way to implement an efficient storage system as conventional log-structured systems do, but it is not sufficient to support the transaction primitives. More importantly, the correctness of the operation is an essential requirement for the primitives. T2 implements a transaction-aware mapping table to address the issue. On the other hand, some flash-based transaction storage systems, such as Deuteronomy [41, 40], take a decomposition approach to separate the data component (DC) and transaction component (TC). While this approach may allow each component to be deployed to various environments, integrating two separate transactions may not be optimal for a highly optimized system. T2's approach is to integrate the higher-level transaction information into the flash-tier subsystem for efficient transactions in order to provide the same correctness of the transaction.

As a transaction-aware mapping table, a mapping table entry contains the address mapping information between the object id and virtual flash offset and two higher-level transaction states: lock state and timestamp information. The lock state and the timestamp information in the mapping table are directly used inside the transaction protocol, and their state change should guarantee the atomicity required by the protocol. The combined data of address mapping information and transaction state represents the current *object state* in the flash tier. The current implementation uses 128 bits for each object state. It uses 64 bits for virtual flash offset and 64 bits for transaction and other meta states.

In the multiple concurrent transaction workload in the datacenter, many requests compete for the same objects. Therefore, the transaction stage change should be lightweight, and the region instances exploit the lock-free state update through *compare-and-swap*

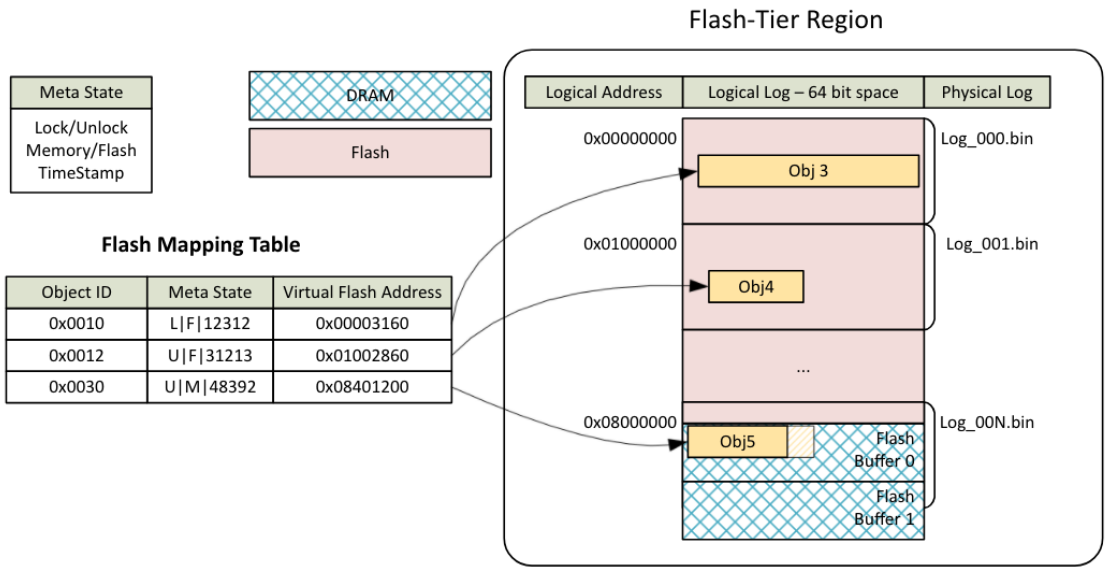


Figure 5.3: A Flash-Tier's Mapping Table and Flush Buffer. The mapping table maps an object id portion of an address to its status, which includes the flash tier's virtual flash offset and meta state; the meta state includes lock status, flash/memory state, and timestamp. An object can exist in DRAM flush buffers or/and in a log file in flash storage. A flush buffer stores an object temporarily in DRAM and the objects in the buffer are flushed into a log file in the background.

(CAS) operations. The current implementation uses Windows operating system's *InterlockedCompareExchange128* [68].

5.2.3 Transaction State Use in Commit Protocol

When a written object is being committed, the transaction coordinator requests a *Lock* operation to a region instance, which updates the lock state atomically. Once it is locked, any operations against the object fail and the corresponding transactions are aborted. Once all the written objects in the same transaction are locked, the transaction coordinator requests *Commit* and sends the current timestamp to the target region instances. Then, each region instance updates the object's timestamp and writes the data into the flush buffer and unlocks the lock state.

When an object is read during the execute phase and its transaction is being committed, the transaction coordinator validates the object's timestamp again to check whether or not the read objects are being modified. When an object is locked or its timestamp is newer

than the transaction coordinator's timestamp, the transaction is aborted. While the read operation during the execute phase involves I/O, its validation is very efficient because it is performed in a lock-free manner.

Unlike the memory tier, all of the transaction state is incorporated into a single 128-bit entry; therefore, this lock-free atomic operation on the transaction state allows the flash-tier subsystem to guarantee the data's integrity regardless of the object size and underlying high-latency flash operations.

5.3 Concurrent Writes On Flush Buffers

As discussed in Section 5.1.1, the local commit phase performance is directly affected by the flash tier's write operation. The flash tier utilizes multiple flush buffers [40] to hide flash I/O latency. Once the object is locked in the transaction state, the object can be written to the flush buffers. This process is entirely performed in memory; therefore, it is very efficient. However, the flush buffers have only limited size, and the region instances need to flush the buffers efficiently. The flush buffers are flushed on demand; if the remaining space in the buffer is insufficient for a new or updated object, it *seals* the current buffer and moves to the next available buffer. The sealed buffer does not accept any changes until it is fully written to the flash storage. Once the data is fully written to the file, the buffer is unsealed and ready to be reused.

5.3.1 Multiple Flush Buffer and States

Each buffer can be in three states to show its current state: current, processing, ready (Figure 5.5). The *current* buffer points to the currently available flush buffer that is being written, and other flush buffers are in the state of *ready* or *processing*. *Ready* flush buffers can be used immediately when the current buffer becomes full, and the *processing* flush buffer is the buffer whose write operations were issued but not completed.

When the current flush buffer is filled, the region instance flushes the buffer and issues

a write I/O to the current log file asynchronously. Then, the next *ready* buffer becomes the *current* flush buffer and serves new write operations. Once the write I/O is issued, the flush buffer becomes *processing* until the write operation completes. The size of the flush buffers is configurable and determined by the workload and its impact on memory buffers.

5.3.2 Concurrent Writes

The flush buffers can aggregate multiple objects into a single flash I/O. This write operation is triggered by *Commit* for a local region or by *Truncate* for a remote region. Although only one lock-guarded object for a specific address can be written during the commit protocol, multiple transactions can request write operations for the same region, which causes high contention. To handle these concurrent updates efficiently, *Write* operation is separated into two steps [40]: atomic reservation and concurrent writes.

The atomic reservation phase is a step in which each write operation reserves its own space, and the concurrent write step allows each thread to write its data concurrently. Figure 5.5 shows the two steps for multiple write operations for the same region. Each thread first requests a reservation of a space through a CAS operation. The flash region instance maintains the starting offset of the current buffer. Each thread retrieves the starting offset and adds its data size to the offset; then it performs a CAS operation to update the starting

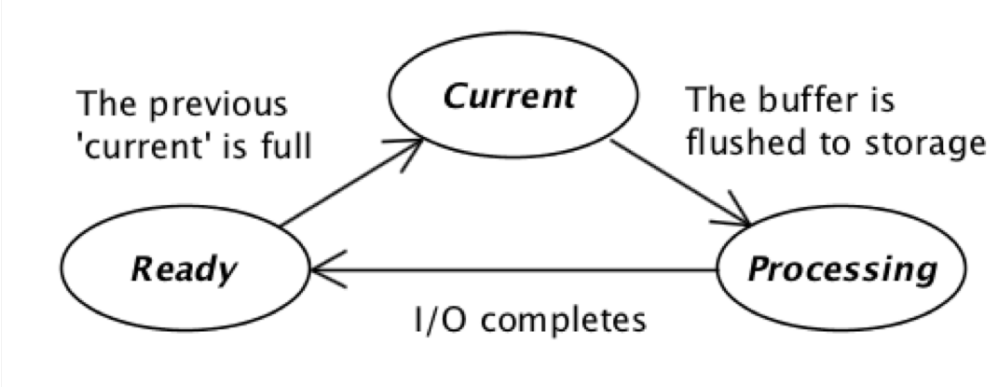


Figure 5.4: Flush Buffer States. The flush buffers have three different states: *current*, *ready*, and *processing*. When a *current* buffer is filled and there is no available space, the flash-tier region flushes the buffer and updates its state as *processing*. Once the data is written to the flash log file, it becomes *ready*.

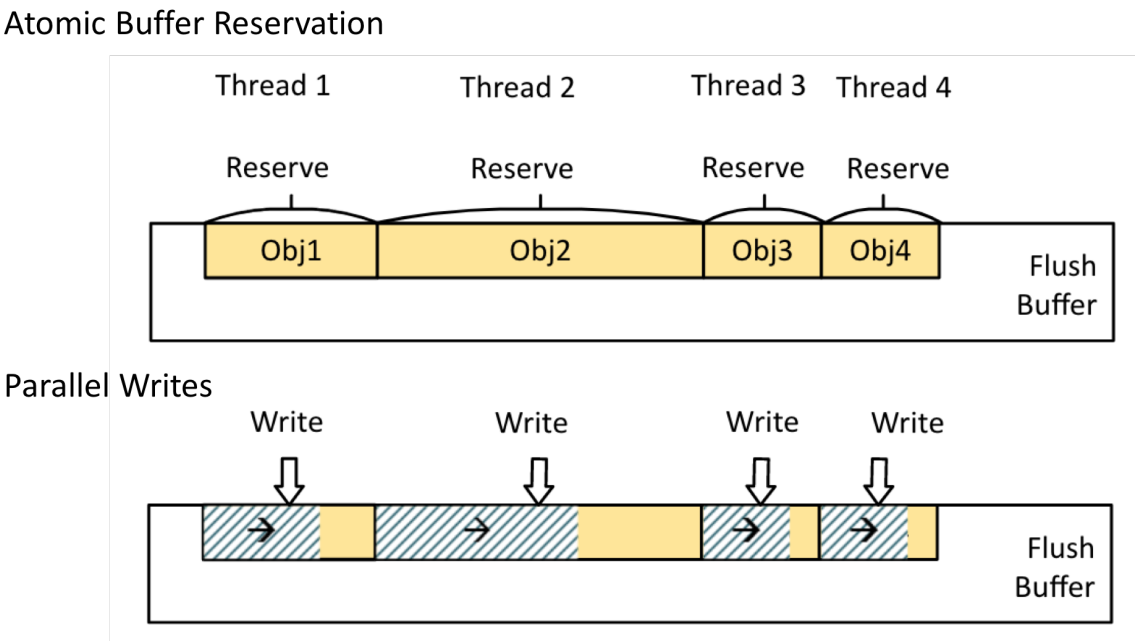


Figure 5.5: Atomic Reservation and Concurrent Writes. Multiple threads simultaneously attempt to reserve their space through a CAS operation. Once the reservation succeeds, each thread can write its update in the flush buffers immediately and concurrently. If the reservation fails, it retries the reservation until it reserves its space.

offset with the new offset. If it succeeds, the thread can perform a write operation, or *memcpy*, to the target object. If it fails to update the offset, it retrieves the current starting offset again and retries the reservation until it succeeds. Because the reservation only needs offset calculation and CAS operations, this atomic reservation is very efficient. Once a thread reserves a space, it can write its object to the reserved space immediately, regardless of other write operations.

5.4 Asynchronous I/Os

The flash-tier subsystem processes all the flash I/O operations asynchronously to improve the overall throughput.

5.4.1 Asynchronous Writes

For write operations, as soon as the space for an object is reserved, the region instance copies the object data into the reserved space and returns. Therefore, the flush buffer can effectively hide any flash-related write latency. The region instance flushes the buffers asynchronously to not block the other transactions. While there is no additional user-level callback function to this flush operation, the region instance updates the flush buffer state from *processing* to *ready* once the I/O completes.

During the reservation, a flash-tier region instance polls the flush buffer state inside a busy loop until it finds a ready buffer. The reason for using the busy loop is to improve the performance in common cases, assuming the conflict between different threads is relatively short. However, this may result in a deadlock. For example, when the workload is too high and the flush cannot be processed fast enough, there are no available buffers for the reservation, and all the threads keep trying CAS to reserve the space without giving any chance for the flush buffer state to be updated.

To prevent such a situation, each region instance should guarantee the availability of the flush buffers by providing more flash buffers than the number of threads, exploiting thread-yielding or throttling write operations. This could be easily solved by configuring the number of flush buffers and the flush buffer size, which are discussed in section 6.2.

5.4.2 Asynchronous Reads

When a datacenter application requests a read operation, in most scenarios, the object is not in the flush buffer but in the log files because the size of the flush buffers is much smaller than the supported data capacity. Therefore, we cannot hide the read latency during the execute phase.

Although the read latency cannot be hidden from a user, the throughput can be improved by processing read requests asynchronously, which is critically important in high concurrent workloads. When a request comes into the flash-tier region instance, it checks

its lock state from the mapping table and issues a flash I/O with a callback function, and then it returns immediately. While the transaction coordinator for the read operation waits for the I/O completion, new transactions can start without being blocked by this read operation. Once the read operation completes and the data is ready in the buffer, the flash-tier region instance calls the callback function.

5.4.3 Asynchronous Operations and Threading Model

FaRM implements its own threading model based on an event loop to improve the performance. Each FaRM thread is pinned to a hardware thread, and each thread polls for different types of events and runs one work item from the event queues ([9]). An I/O event is one of the event types that FaRM thread is polling, and each event should be handled by the thread that issues the request. Therefore, each region instance cannot simply issue asynchronous *Read* operations and return.

In this threading model, *Read* I/O should be handled by the same thread it issues because the user-level data structure is still required to complete the *Read* operation. Therefore, the asynchronous read operation involves thread-message techniques to overcome this constraint when a requester is running on a different thread from the region instance.

Write does not have this constraint because no user thread-level data structure is involved. The buffer state update is always processed in the same thread that issues the flush requests.

5.5 Implementation

We implemented the flash-tier subsystem of T2 in C++ and extended FaRM's transaction protocol to handle both memory and flash tiers. T2 is running on Windows systems, which use the NTFS file system. We configure file operations not to use the file system's caching mechanism so that T2 can explicitly control the flash tier's buffering scheme, as discussed in section 3.4.

5.6 Summary

In this chapter, we reviewed the commit protocol and considered the performance and correctness of the flash-tier integration. We then described the techniques to improve the performance and guarantee the transaction correctness for the flash-tier subsystem. We integrated the transaction state into the mapping table and utilized the CAS operations to update the transaction status and data atomically and efficiently. This prevents the introduction of the flash tier from changing the existing transaction semantics. We also exploited the atomic buffer reservation and concurrent write techniques to reduce the write overhead and also use asynchronous I/Os to improve overall throughput.

There are several potential topics that can enhance performance of the flash-tier subsystem, such as Multi-Version Concurrency Control (MVCC) and caching for the flash-tier systems. These topics are addressed in section 7.2 as future research.

CHAPTER 6
PERFORMANCE EVALUATION

The flash-tier subsystem of T2 not only preserves the simple programming model but also provides efficient transaction through a transaction-aware mapping table, compare-and-swap state updates, and asynchronous operations. This chapter quantitatively evaluates the performance of an individual flash-tier region instance and of T2 as a system with a set of custom micro-benchmarks and a commonly used cloud-serving benchmark, YCSB [31]. The chapter then demonstrates the cost effectiveness of the flash tier, which can provide the competitive or better throughput for the same price compared to the memory tier.

Before delving into the analysis of cost effectiveness, it is useful to understand the performance characteristics of primitive operations from the two levels: flash-tier region instance and protocol of T2. The first experiments utilize two custom micro-benchmark suites: FlashRegionBench and TxBench. FlashRegionBench measures the performance of a region instance's primitives, such as *Read* and *Write*, and TxBench measures the performance of the individual transaction APIs, such as *TxRead* and *TxCommit* and the overall performance of the flash-tier subsystem in T2. Details are provided in section 6.2.

6.1 Experiment Setup

The experimental testbed consists of 10 machines running in an isolated environment. Each machine has 256 GB of DRAM. Each has two 8-core Intel E5-2560 CPUs, and each runs Windows Server 2016 Datacenter. We enabled hyper-threading (total of 32 hardware threads per machine) and used 6G HP 200GB SSDs (HP MK0200GCTYV) for flash storage. For the RDMA message communication, each machine has two Mellanox ConnectX-3 56Gbps NICs. The capacity for the memory and flash regions are configurable and set to 20 regions for the memory tier and up to 32 regions for the flash tier. We set the region size

to 2GB for both tiers.

6.2 Micro Benchmark - FlashRegionBench

FlashRegionBench is a custom micro-benchmark to evaluate the performance of the primitive operations of a flash-tier region instance for various system configurations. As a building block for the commit protocol, the flash-tier region instance provides the primitive APIs to support transaction operations (Table 5.1). In particular, this section focuses on the *Read* and *Write* operations, which directly affect the overall transaction performance in the course of the commit protocol. The other APIs, *Allocate*, *Free*, *Validate*, and *Lock*, are the state change operations in the in-memory mapping table; therefore, those operations are quick and/or are used with the *Read* or *Write* operations. Therefore, the performance of the flash-tier region's *Read* and *Write* operations are discussed in this benchmark.

6.2.1 Description of FlashRegionBench

FlashRegionBench is devised to measure the baseline performance of one region instance for *Read* and *Write* operations in various configurations. It creates a flash-tier region instance on a target machine and the region instance is pinned to a physical core. Multiple workers run on the same machine, but they are running on different threads. The workers issue *Read* and *Write* requests to the target flash-tier region instance during the experiment. To increase the level of concurrency, two types of concurrency can be utilized. First, each worker is mapped to a physical core, which gives the thread-level concurrency. Second, a worker can issues multiple requests while the thread of the work awaits the completeness of any pending I/Os, which gives the user-level concurrency.

The experiments are conducted in three stages: initialization and warm-up, experimentation, and cool-down. The first initialization and warm-up stage *Allocate*s objects in the target flash region, updates the objects with arbitrary data, and *Write*s them. The addresses of the allocated objects are kept in the global data structure, and each worker thread ran-

domly picks one or more objects from these addresses and requests primitive operations in the experimentation phase.

During the experimentation phase, each worker issues *Read* or *Write* operations. While the *Read* operation can be performed without any dependency as long as the target object is not locked, the *Write* operation requires that *Lock* be performed successfully in advance; if *Lock* fails, the benchmark continues with a different object. The number of worker threads and the level of user-level concurrency are configurable; therefore, multiple workers are concurrently running and issue multiple *Read*s and *Write*s simultaneously.

Finally, the cool-down phase waits for the completion of experimentation and returns the collected statistics.

6.2.2 Benchmark Configuration

We ran FlashRegionBench for 60 seconds for various system configurations and collected the performance statistics. The region size of the flash-tier region is 2 GB, and the size of each object is 1 KB. The region is fully allocated by the objects, and the number of total objects is approximately 2 million. The number of flush buffers and the flush buffer size are configurable.

Table 6.1: FlashRegionBench Configuration.

Region Size	Total Buffer Size	Object Size	Threads	User-level Concurrency
2 GB	2 MB	1 KB	1 to 32	1 to 32

6.2.3 Benchmark Result

Write

Figure 6.1 shows the *Write* throughput of a region instance. The throughput increases as the concurrency level increases in both thread- and user-levels; however, there is the maximum throughput that the flash-tier region instance can achieve due to the flash storage

I/O bottleneck. In an ideal case , where there is no overhead in the flash-tier region instance, the maximum throughput is the sequential write throughput to a single file because T2 utilizes the log-structured way of writing the flush buffers.

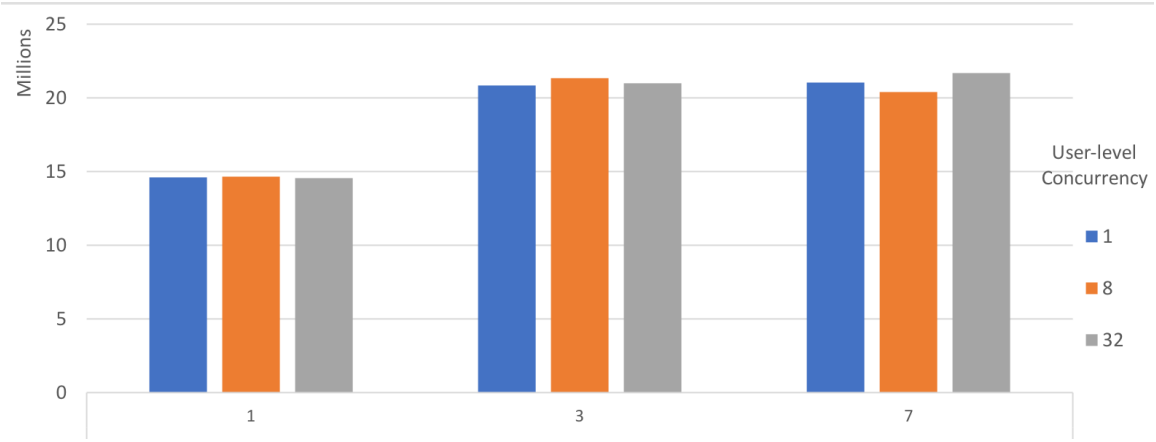


Figure 6.1: Region Instance *Write* Throughput. The graph shows the *Write* throughput of a single region instance. The y axis represents the number of *Write* operations and the x axis represents the number of worker threads. Blue, orange, and gray bars represent the user-level concurrency. Twenty million 1 KB *Write* operations for 60 seconds in the experiment is equivalent to 317 MB/s. As concurrency degree increases, the throughput increases and is saturated, which is equivalent to 96% of the maximum throughput, and 364 MB/s in our sequential file write test.

To evaluate the *Write* performance, we also measured the file system's I/O performance for sequential writes via Windows system's WriteFile API. When we measured the write throughput of WriteFile, the target file was created with the following options: asynchronous, no buffering, and write-through [69], which are the same for the flash-tier region instance.

Our experiment shows that a single flash-tier region instance can efficiently process the *Write* operations and achieve 96% performance of the maximum sequential file write throughput, which shows that the overhead of *Write* is negligible. This allows the commit phase in the commit protocol to be processed very quickly, as shown in section 6.2

Read

In contrast, *Read* operations on flash-tier regions are much slower than *Write* operations. Unlike the buffered write operations, read operations need to access the corresponding files

directly; further more, the object size is relatively small and its distribution is random. Therefore, *Read* operations expose the underlying file system's high read latency directly to users which is the dominant factor of the overall transaction, as discussed in the next section.

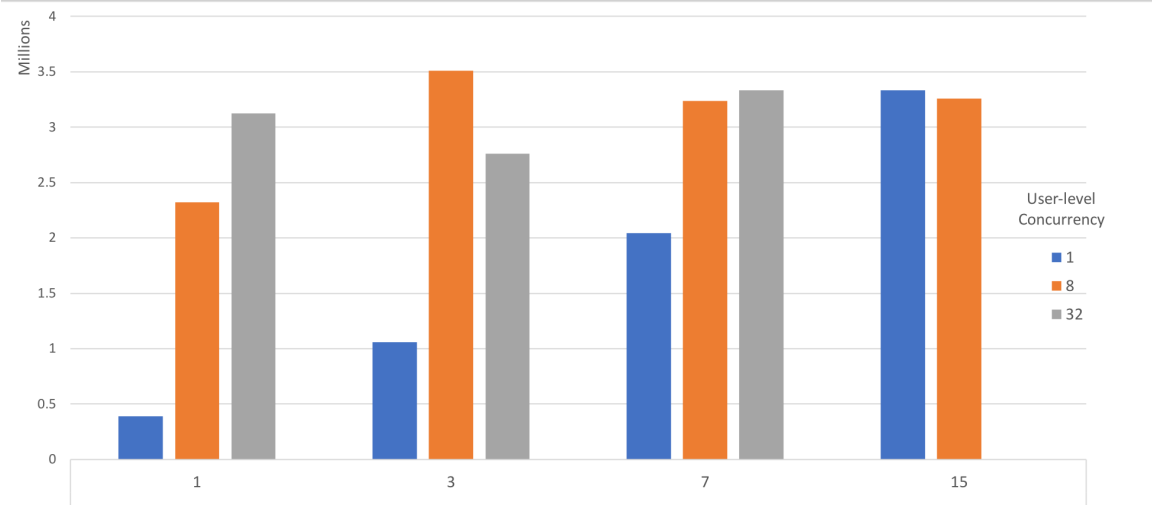


Figure 6.2: Region Instance *Read* Throughput. The graph shows the *Read* throughput of a single region instance; the legends are the same as those in Figure 6.1. Unlike the *Write* operations, *Read* cannot hide its high I/O latency from its data path; furthermore, its access pattern is random by its nature. On the other hand, the read operation is implemented as asynchronous; therefore, the overall throughput is improved as the concurrency increases, either thread-level or user-level. However, its maximum throughput is also saturated by the underlying flash storage's I/O performance.

This section covered the primitive operations provided by a flash-tier region instance and shows its performance and limitation. In the following section, the higher-level primitive transaction in T2 will be discussed. TxBench will be used to analyze the performance of the primitive transactional operations that are implemented using *Read* and *Write*.

6.3 Micro Benchmark - TxBench

TxBench is a custom micro-benchmark to evaluate the performance of the primitive transactional operations on T2 for various system configurations. While FlashRegionBench measures the performance of the primitive operations of a flash-tier region instance, TxBench measures the performance of a set of transactional primitives wrapped in a transaction context. TxBench concurrently issues a set of primitive transactional operations against T2;

it then gives performance metrics and statistics, such as, throughput, latency, statistics of individual operations, events, etc. While the workloads of TxBench are simpler than those of real-world workloads, these workloads show the baseline performance of the T2 system.

6.3.1 Description of TxBench

A benchmark worker is a running thread that generates a workload, and workers of the benchmark on a separate machine issue a set of read or/and write operations within a transaction context concurrently for an experiment duration. Each worker is mapped to a physical thread and running on an individual core; therefore, multiple workers give the thread-level concurrency. This experiment does not utilize the user-level concurrency because there is no additional performance improvement when the number of threads is more than or equal to eight.

The experiment is conducted similarly to the FlashRegionBench method. The first initialization and warm-up stage populates objects in the flash tier and the memory tier. The workers allocate a target number of objects, write data to the objects with arbitrary data, and commits them. The objects are evenly distributed on testing machines and their addresses are kept in each worker; then, each worker randomly picks one or more objects from these addresses and issues requests in the next experiment phase.

Table 6.2: TxBench Workload.

Pattern	**Wa**	**WaRb**
Description	Read and write an object	Read and write an object, and read another object

During the experimentation phase, each worker issues predefined sets of read and/or write operations (Table 6.2), and each set of operations is performed in a single transaction context. The sequence of each transaction consists of the following primitives: 1) creating a transaction context, 2) performing the predefined sets of operations (Wa/WaRb), and 3) committing the transaction. A transaction is aborted if there is any conflict; in this case, the worker reissues the failed request. Multiple workers are concurrently running and issue

multiple transactions simultaneously.

6.3.2 Benchmark Configuration

We loaded TxBench with one million objects per machine before running transaction operations. The size of each object is 2 KB, and the objects are evenly distributed among the threads. We used two 2 MB flush buffers. Each thread has its dedicated flash-tier region instance, and each flash-tier region is created during the initialization. The worker threads are also evenly distributed in a machine. We disabled several optimization settings, such as cache, so that we can measure the actual performance of the underlying tiers.

We ran TxBench for 60 seconds for various system configurations and workloads and collected the performance statistics.

We first analyzed the performance of T2 running on a single server, which gives the baseline performance of transaction APIs. Subsequently, we moved to a nine-machine configuration. The first single-machine configuration is used to analyze the consequence of T2's design at the level of transactional APIs, and the nine-machine configuration is used to understand the overall impact of the flash tier.

6.3.3 Benchmark Result

We first look at the head-to-head performance comparison between the memory and flash tiers in a single-machine configuration.

Transaction Performance Comparison

Figure 6.3 shows the throughput comparison between the memory and flash tiers for two simple transaction workloads.

The two types of transaction workloads are chosen to analyze the primitive transaction operations. Workload *Wa* is a single transaction workload that reads an object, *a*, updates it, and commits it. Workload *WaRb* is another single transaction workload that reads and

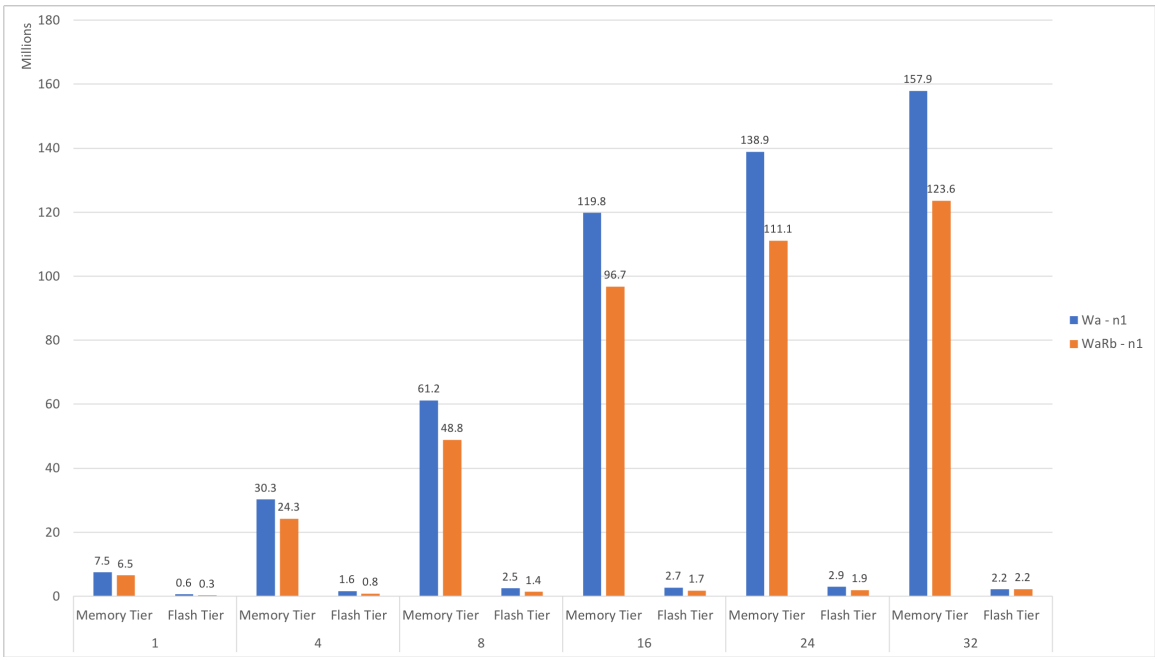


Figure 6.3: Transaction Throughput Comparison. This shows the head-to-head throughput comparison between the memory and flash tiers in a single-machine configuration. The x axis represents the number of threads, and the y axis represents the throughput, or transactions per second, for two types of transaction workloads: Wa and WaRb.

updates an object, *a*, and then reads another object, *b*. Workload Wa is the simplest form of transaction that exercises the full commit protocol, and the user-level latency of the transaction is measured from *TxCreate* to *ACK* (Figure 5.1); *TxTruncate* is a background operation whose performance is hidden to users. Workload WaRb adds an additional read to Workload Wa, and it shows the impact by an additional read operation in the same transaction context.

First, the mere difference of transaction throughput between the memory tier and the flash tier is huge. The throughput of the memory tier is up to two orders of magnitude higher than that of the flash tier in high concurrency, i.e, the throughput of the memory tier with 32 worker threads is 72 times that of the flash tier for workload Wa. The considerable performance difference in the head-to-head comparison is expected considering T2's design constraints; T2 excludes the use of cache for any read latency improvement so that it can provide more memory to users.

Looking into the individual operation shows the performance impact by *Read* and *Write*

operations. The main bottleneck that limits the flash tier's performance is the read latency during the transaction execution phase, and the read latency of the flash tier takes up to 92 % of its total transaction latency.

Transaction Primitive Performance

We measured the performance of the individual transaction primitive operations of T2 to understand the performance bottleneck. A measured transaction is from workload Wa and consists of the following transaction primitives: *TxCreate*, *TxRead*, *TxLock*, *TxWrite*, and *TxCommit* (Figure 4.2). As discussed in section 5.1, *TxRead* takes the major part of the execution phase, and the duration between the time to request *TxCommit* and the time users get ACK takes the major part during the commit phase. The other primitives are negligible compared to *TxRead* and *TxCommit*.

For the memory tier, these two phases take the same order of magnitude; however, for the flash tier, the execution phase dominates the latency for each transaction and affects the overall performance.

Figure 6.4 shows the latency of two transaction primitives, *TxRead* and *TxCommit*, inside a single transaction. For the memory tier, the TxRead latency is very low and both TxRead and TxCommit latencies have a similar impact on the overall performance. However, for the flash tier, the TxRead latency takes the significant portion of the transaction, 80% to 92% in the experiment, of the overall latency.

Specifically, the TxRead latency of the flash tier is two orders of magnitude higher than that of the memory tier (Figure 6.6), which is consistent with the known latency difference between memory and flash storage (Table 3.4). While the latency increases of the memory tier is negligible as the number of worker threads increase, the latency increase of the flash tier is noticeable in the high concurrency workloads, or more threads.

On the other hand, our measurements show that the TxCommit latency of the flash tier is just 50% to 83% higher than that of the memory tier depending on the degree of

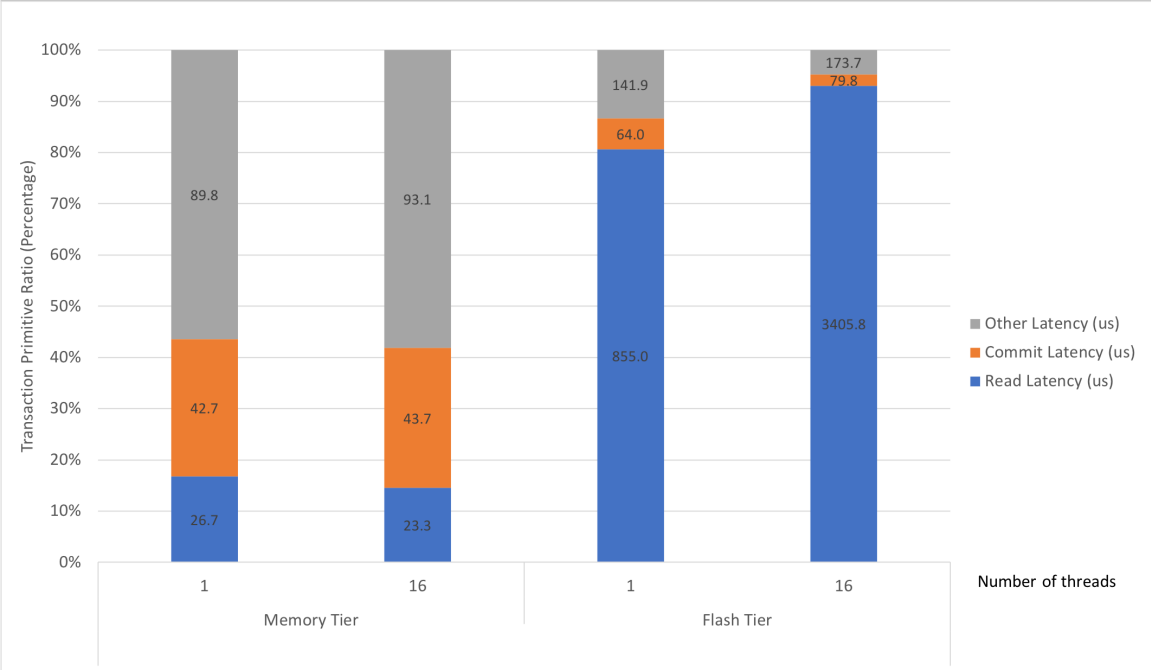


Figure 6.4: Latency Ratio of TxRead and TxCommit. The graph depicts the relative ratio of the latencies of TxRead and TxCommit in a single TxBench transaction in T2's memory and flash tiers. The other latency includes TxLock and TxBench's request preparation step. The flash tier's overall transaction time is dominated by the TxRead latency because an object is directly read from the flash storage.

concurrency.

Throughput vs. Latency

The throughput of a system is not independent of the latency. As the concurrency increases, the throughput accordingly increases; however, this also causes high contention among the threads to reserve a space in the flush buffers and to use I/O resources.

Figure 6.7 shows the relation of throughput and latency of the flash tier for workload Wa of the TxBench in the experiment described above (Figure 6.3). The throughput increase rate decreases significantly as the level of concurrency increases, and the throughput starts to decrease at the high concurrency condition. On the other hand, the latency keeps increasing at a constant rate. Consequently, it is critical for users to understand their workloads and find an appropriate *sweet spot* for the workload. In the following experiments, we chose eight threads as a sweet spot configuration for the flash tier, which achieves 85 %

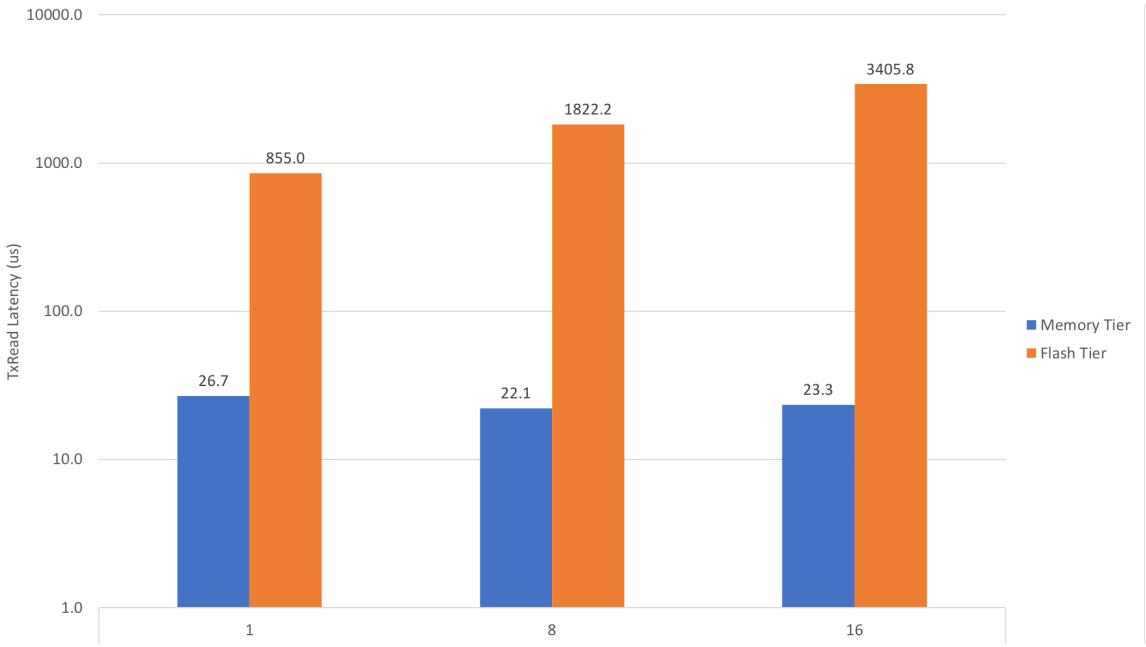


Figure 6.5: TxRead Latency. Flash tier's TxRead latency is two orders of magnitude higher than memory tier's latency, which results in the huge performance differences in the overall commit performance. The x axis represents a number of threads, and the y axis represents the read latency. As the number of threads increases, the latency also increases due to the contention for I/O resources; the throughput also increases though this is not shown in the graph. Note that the scale of the y axis, latency (us), is the log scale.

of the maximum throughput with the 2.1x minimum latency.

Scalability and Replication

Finally, Figure 6.8 plots the throughput and the latency for the Wa type, varying the number of servers and replication degree. The results show that the flash-tier's region performance scales well with the cluster size. The baseline for the three-machine performance achieves 2.8 million requests and the nine-machine performance does 6 million and 5.5 million requests for a single replica and three replica configurations. The throughput increases 2.2x and 2.8x respectively. In a multiple-machine configuration, remote object access is always performed through a regular RPC operation (section 5.1) rather than an efficient one-sided RDMA.

Compared to a single replica performance, the overhead of three replicas for the backup commits is about 8.7% of the performance at the nine-machine configuration.

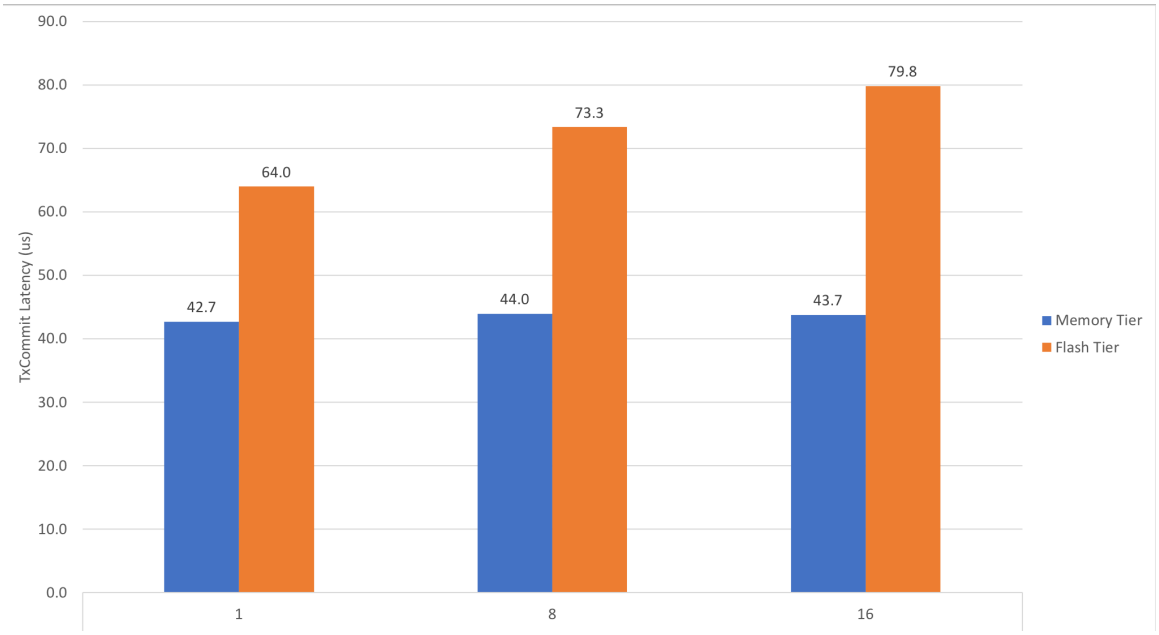


Figure 6.6: TxCommit Latency. Flash tier's TxCommit latency is just 1.5 to 2 times higher than memory tier's latency. The x axis represents a number of threads, and the y axis represents the commit latency. As the number of the threads increases, the throughput and the latency both increase. Note that the scale of the y axis, latency (us), is the linear scale.

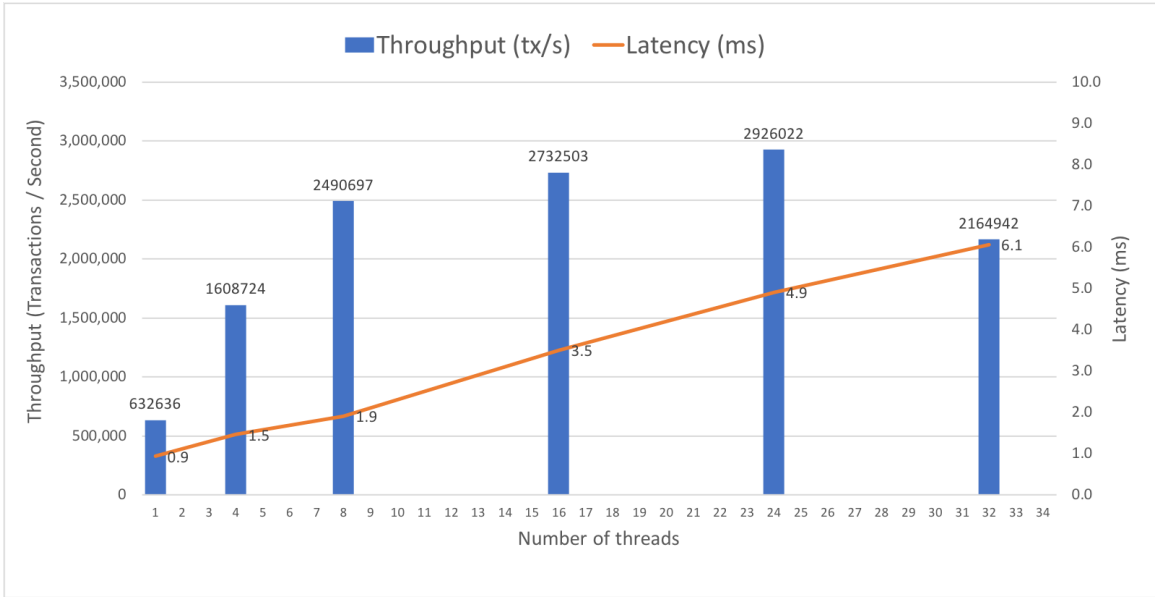


Figure 6.7: Flash Tier's Throughput vs. Latency. This shows the throughput and latency of the flash tier for workload Wa in the above TxBench experiment. While the throughput increase rate decreases and even becomes negative, the latency keeps increasing at an almost constant rate. In our experiment, the sweet spot is the eight-thread configuration.

When both read and write operations are involved, the scalability is limited by both the regular RPC communication and I/O read latency because we need to validate the ob-

ject during the commit phase. In contrast, read-only operations do not exercise the full commit protocol. For example, for the flash region objects, T2 needs to validate the read through a regular RPC mechanism, which adds additional CPU overhead and introduces performance degradation in both communication and CPUs.Therefore, due to the lack of RDMA and high read latency in a flash-tier subsystem, the scalability of the complex types of transactions is limited.

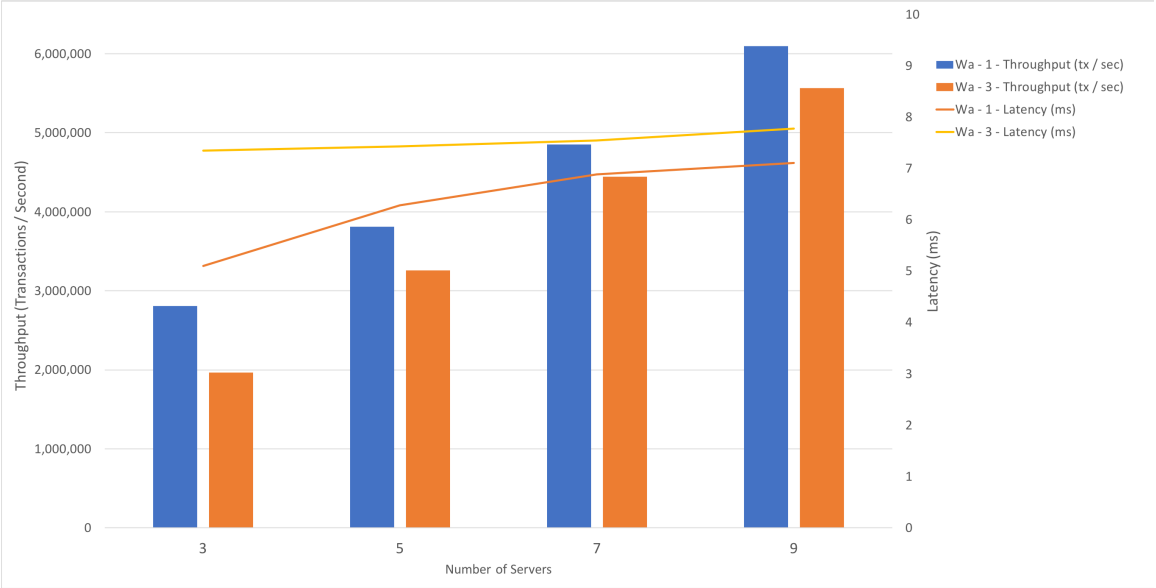


Figure 6.8: Scalability and Replication. This graph shows the performance of the flash tier for the scalability and replication. The test runs on a different number of servers, and each instance runs eight threads with one-replica and three-replica configurations. The x axis represents the number of servers and the y axis represents the throughput (left) and the latency (right). For comparison, three-server configuration is chosen because it exercises the full commit protocol, including the backup replication. Replication overhead for the flash tier is 8.7% with a nine-server configuration. Compared to the performance of three-server configuration, the throughput has increased by 2.2x (1 replica) and by 2.8x (3 replicas) at the nine-server configuration.

Up to now, we have examined the flash-tier's performance characteristics in various configurations, from the simplest single-server case to the practical nine-server configuration with three replicas. In the discussion below, the cost effectiveness of the flash tier is considered based on cost and throughput in the context of TxBench and YCSB.

6.3.4 Cost Effectiveness

Per-Dollar-Throughput

Cost effectiveness is a broad concept; therefore, we first need to limit our discussion by defining what it means as a way to analyze cost effectiveness quantitatively. We evaluate cost effectiveness by evaluating *per-dollar-throughput*, which is the throughput (ops/sec) per unit capacity cost, or dollar per GB (Table 3.2). For example, if a system spends five dollars for the 1 GB memory tier to achieve five-million transactions, can the 1 GB flash tier costing one dollar achieve at least one-million or more transactions? This measures how *effectively* a system can provide the throughput for the *cost*.

Performance Comparison

Figure 6.9 shows the head-to-head performance comparison in the eight-server and three replica configuration. The memory tier uses 12 threads and the flash tier uses 8 threads as a sweet spot configuration.

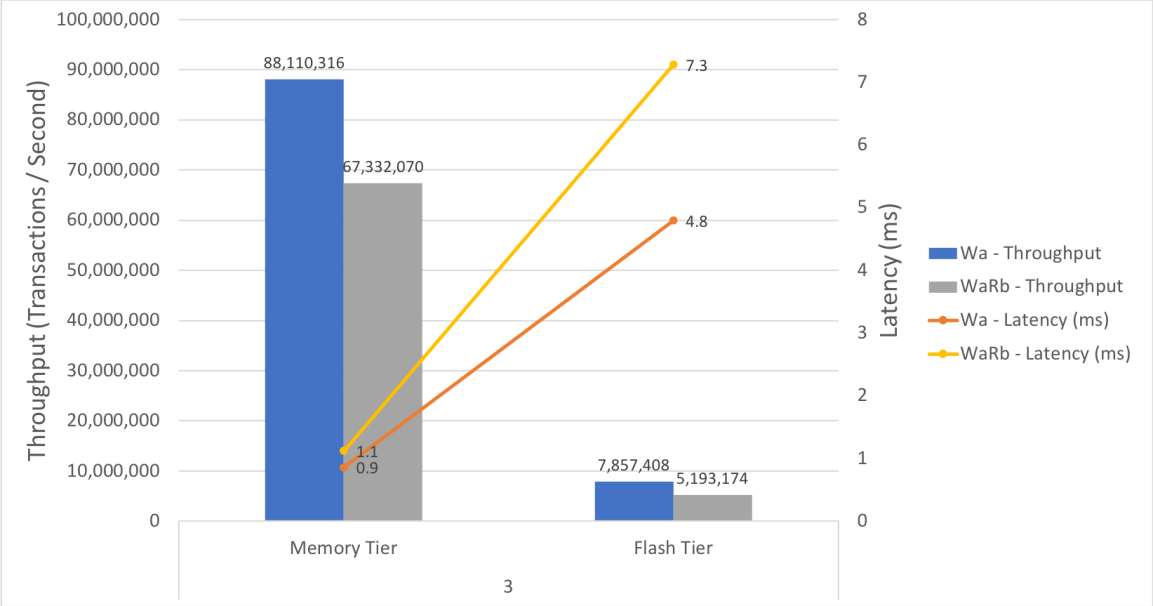


Figure 6.9: TxBench Throughput and Latency Comparison. This graph shows the head-to-head throughput and latency comparison of the memory and flash tiers for the eight-server and three replica configuration.

The throughputs of the flash tier for the two workloads Wa and WaRb are 8.9% and 7.7% of those of the memory tier and the latencies are increased 5.3 times and 6.6 times respectively. The performance degradation in the flash tier was expected and the experiment shows that the flash tier performs inferior to the memory tier in the primitive transaction workloads.

However, the level of degradation in the flash tier does in fact demonstrate the cost effectiveness of the system when its cost and performance are considered together.

Cost Effectiveness in TxBench

In Table 3.2, we compared the cost per GB of commodity DRAM and SSD, and DRAM cost is 23 times higher than SSD cost. Combining the cost with the above performance, Figure 6.10 shows the throughput per dollar, or *per-dollar-throughput*, for the memory tier and flash tier.

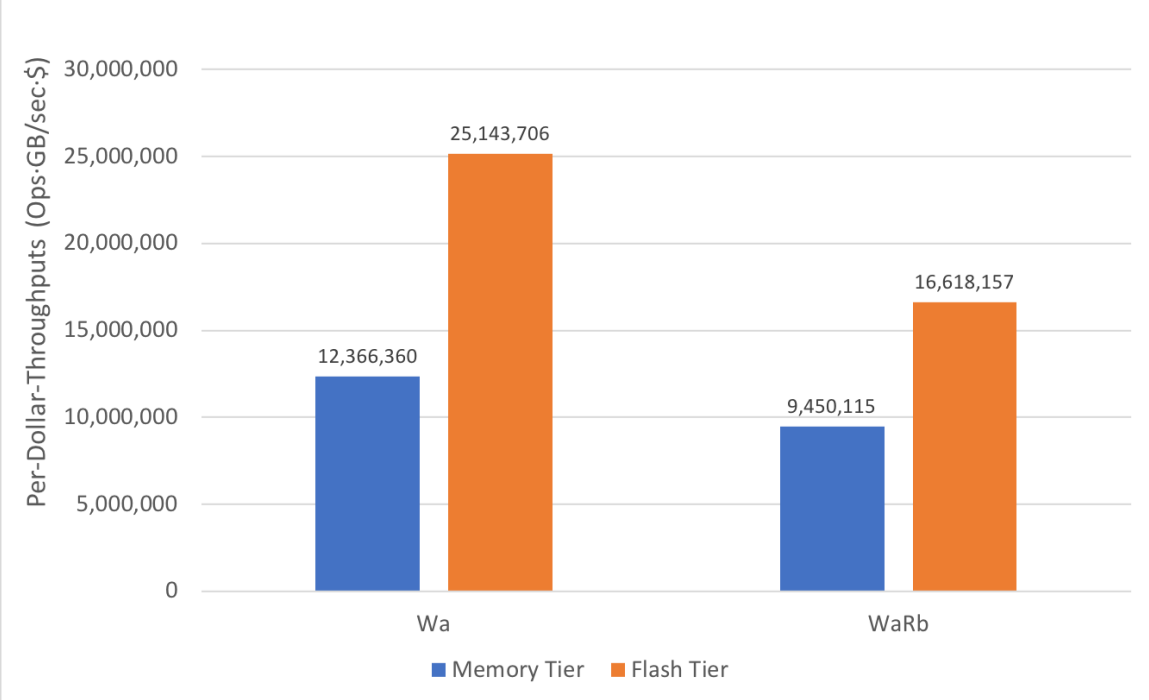


Figure 6.10: Per-Dollar-Throughput in TxBench. This diagrams shows the *per-dollar-throughput* of the memory tier and flash tier based on the commodity prices shown in Table 3.2. The *per-dollar-throughput* of the flash tier is 203.3 % and 175.9 % of the memory tier for workload Wa and WaRb.

In TxBench with two primitive transaction workloads, the flash tier's per-dollar-throughput is 203.3% and 175.9%of the memory tier for the same workload Wa and WaRb. Consequently, we can see that the performance of the flash tier is cost effective in throughput.

On the other hand, latency cannot be effectively improved without employing cache and sophisticated eviction policies. Therefore, a latency sensitive application may not utilize the larger capacity through flash storage directly. Instead, if the application can separate a latency-sensitive part and latency-not-sensitive part, and the modification of the two parts can be in a different transaction, then the application can utilize cost effectiveness of the system.

In the next section, we run YCSB to show the flash tier's performance and its cost effectiveness in a realistic context.

6.4 YCSB

YCSB is a widely used benchmark for cloud-serving systems [31]. In this dissertation, there are two implementation differences from the common YCSB setting. First, while the common YCSB benchmark measures the performance of a non-transactional workload of CRUD operations, our benchmark measures the performance of a transactional workload. Each operation is performed in a transaction context; it succeeds if its data is transactionally committed and can fail if there are any conflicts. Second, our YCSB benchmark uses a B-tree index on the memory tier to implement key-value APIs. The memory B-tree maps a key to an address that points to a location storing its value, which is a common practice in industry; hot, small, and frequent meta-data is stored in memory and cold, large, and infrequent user-data is in flash. While we also measured the performance of YCSB implemented with only the flash tier including the B-tree data structure, its performance was very inferior to that of the memory tier and is not useful in memory-flash hybrid systems, so it is not included in this dissertation.

6.4.1 Performance Comparison

Figure 6.11 shows the throughput of the memory tier and the flash tier at the nine-machine configuration. Each key was chosen randomly with a uniform distribution, varying the ratio of read and update (Table 6.3).

Table 6.3: YCSB Workload.

Workload	au	bu	cu
Description	R 50%, U 50%	R 95%, U 5%	R 100%, U 0%

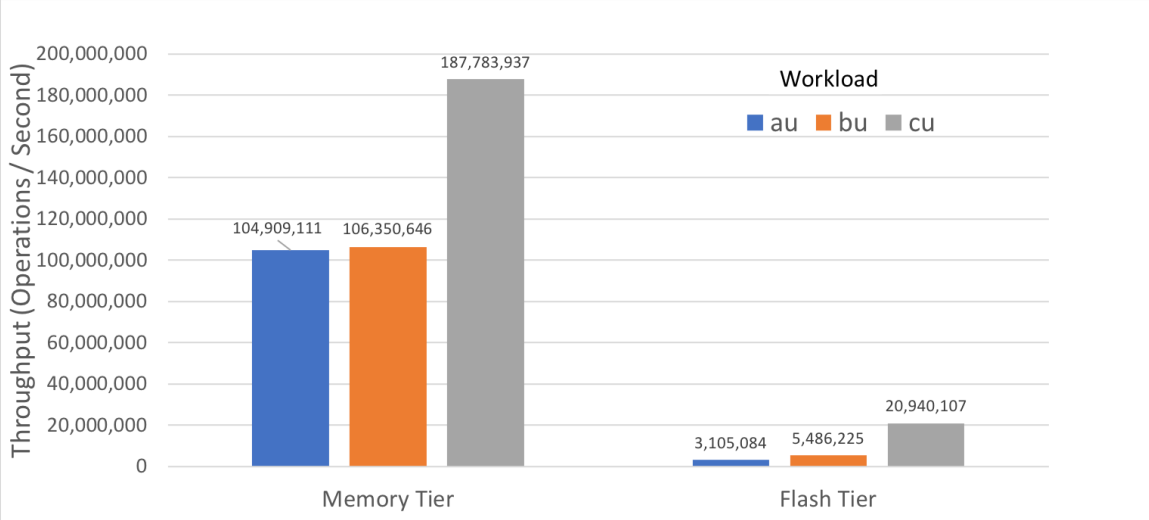


Figure 6.11: YCSB Throughput Comparison. This test runs YCSB on the memory tier and the flash tier, and each instance runs eight threads with three replica configurations. The y axis represents the throughput of the transactional CRUD operations. The throughput of the flash tier is 3.0% (au), 5.2% (bu), and 11.2 % (cu) compared to that of the memory tier.

The performance of the flash tier is 3.0% (au), 5.2% (bu), and 11.2 % (cu) of that of the memory tier. The relative performance differs based on its workload. As the read ratio increases, the throughput increases because the *read-only* transaction does not need to take the entire commit protocol. Once the data is read at the time of transaction start, it is valid at that time and does not need to communicate with other replicas again. The flash tier's performance improvement (6.7x) of workload cu over au is much higher than that of the memory tier (1.7x). For a workload containing updates, the entire commit protocol is performed and it introduces high-latency regular RPC calls for validation and commit

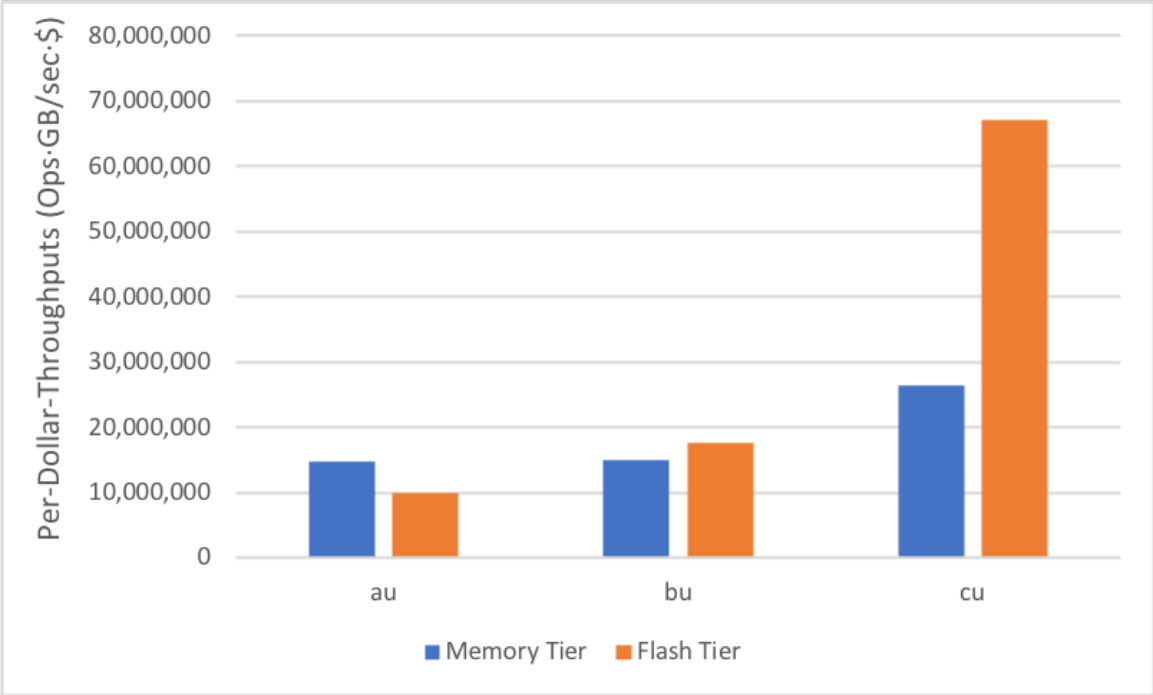


Figure 6.12: Per-Dollar-Throughput in YCSB. This diagram shows the per-dollar-throughput of the memory tier and flash tier calculated with the commodity prices shown in Table 3.2. The per-dollar-throughput of the flash tier is 67.5% (au), 117.6% (bu), and 254.2 % (cu) of that of the memory tier.

for the flash tier, which increases more latency than the one-side RDMA way of validation and commit for the memory tier. Therefore, the flash tier is better suited for a *read-only* workload than for a mixed workload.

The performance degradation in adopting the flash tier was expected and the experiment result shows that the flash tier's performance is inferior to the memory tier in a real world workload. However, this benchmark result also demonstrates that the flash tier is cost effective when its cost and performance are considered together.

6.4.2 Cost Effectiveness

Figure 6.12 shows the *per-dollar-throughput* for the memory tier and flash tier with the three replica configuration with the same price configuration as in section 6.3. For a high-update workload (au), the flash tier's per-dollar-throughput is 67.5% of the memory tier; however, as the read ratio increases, the per-dollar-cost of the flash tier is competitive (bu,

117%) and even better for the read-only workload (cu, 254%).

Where the latency is concerned, the flash tier may not be a way to achieve cost effectiveness; however, when the latency is not in the critical path but the throughput is important, the flash tier can effectively improve cost effectiveness–in particular, when the workload is read-oriented.

6.5 Preserving a Simple Programming Model

The experiments in this chapter showed that the flash tier is cost effective in both TxBench and YCSB, in particular for read-oriented workloads. Having showed the cost effectiveness of T2, this section briefly discusses the other design goal of T2, which is to preserve the simple programming model.

For big data applications, it would be cost prohibitive to fit the entire data set in DRAM. Therefore, a tiered storage system (DRAM, flash, and disk) is a realistic necessity for such big data applications. In the absence of a unified programming model such as T2, which encompasses both DRAM and flash tiers, developers would be forced to think of splitting the dataset between these tiers on their own. Further, they would have to deal with the additional complexity of inventing and incorporating the transactional and consistency semantics for their datasets that span these multiple tiers. Programming models such Map-Reduce [70], Spark [71], and Persistent Temporal Streams (PTS) [72] are examples of systems that simplify the programming model to aid the developers and do all the heavy lifting under the covers in the runtime[1].

In a similar vein, T2 provides a seamless programming model for big data applications that would necessarily span DRAM and flash tiers by incorporating the transactional and consistency complexities, and providing a uniform API that spans both tiers (Figure 4.2) and the associated transaction-aware mapping table.

[1]Map-Reduce and Spark hide the complexities of using massive parallelism for embarrassingly parallel applications such as web searches and other throughput-oriented datacenter applications, while PTS provides a seamless API for accessing in-memory and archival data for live streaming applications.

For example, TxBench and YCSB are *applications* of T2, which have been used to evaluate memory-only FaRM's performance. When we extended TxBench and YCSB to run against both the memory tier and the flash tier, the only change that was needed in the benchmark applications was setting storage attribute to a target tier.

Needless to say, application developers need to change application logic regarding how to use the memory and flash tiers based on their workload expectations. However, the simple programming model preserved by T2 allows application developers to tackle big data workloads correctly and with agility.

6.6 Summary

This chapter discusses the evaluation of the performance characteristics of T2. We devised two custom benchmark suites, FlashRegionBench and TxBench to understand the performance characteristics of the flash-tier region instance and of primitive transaction API by the flash tier. The flash-tier's overall transaction is dominated by *txRead* even if the performance of *txWrite* is as efficient as that of the memory tier. The throughput increases as the degree of concurrency increases up to certain levels of concurrency; however, due to the interaction between multiple concurrent operations, the throughput is saturated or degrades after a certain level of concurrency. For simple read/write transaction operations, the throughput of T2 increases proportionally as the number of servers increases; however, when more than one server is involved, the throughput improvement is limited because it cannot exploit one-sided RDMA operations for validate and commit and involves a regular RPC communication during the commit phase. The YCSB experiment shows that while the flash tier's sole throughput is inferior to that of the memory tier, flash tier's throughput per dollar is competitive or better than the memory tier when the workload is read-oriented, so the flash tier provides a cost-effective solution. Finally, because of the simple programming model preserved by T2, the only necessary change of the benchmark applications to tackle both memory and flash tiers is setting storage attribute to a target tier.

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

Designing a cost-effective storage system has been the long-recurring theme in systems research, and it is still valid in the context of modern in-memory distributed storage systems that provide high performance and ease of programming at the same time. Memory is still too expensive to embrace the ever-increasing demand for *data-intensive* applications and as a result, industry keeps seeking a way to reduce the cost. Leveraging cheaper storage is a natural way to resolve such needs in a cost-effective manner; however, achieving cost effectiveness without compromising the simple programming model offered by the new in-memory systems is not trivial.

FaRM, a high-performance distributed in-memory storage system that provides both high performance and ease of programming with ACID transaction, is an example of such in-memory systems, and the request for cost effectiveness is repeatedly asked, just as before. This poses a question: How should we architecture a tiered transaction storage system, such that it can leverage the cheaper flash storage to support the large data demand effectively *and* preserve the simple programming model?

T2 is the *T*iered *T*ransaction storage system to answer to that question. To address the ease of programming, T2's flash-tier subsystem provides *memory-compatible* interfaces through a *region instance*, which creates an illusion of transactional memory out of non-transactional commodity flash storage. An object in the flash address space is modeled to be *visible* to users and *static* once it is allocated. The object is identified by its unique *id*, and flash-tier objects are managed at the granularity of *region* by the region instance.

To address the effectiveness or high performance, T2 incorporates the higher-level

transaction state into the *transaction-aware mapping table*. The region instance performs CAS operations against the object's transaction state to implement atomic transactional primitives. This allows the flash tier to provide efficient transaction primitives and to preserve the correctness of the existing transaction protocol. In addition, it utilizes the atomic buffer reservation technique to allow concurrent writes on the multiple flush buffers. All the flash I/Os are processed asynchronously to increase throughput and prevent blocking.

7.2 Future Work

T2's flash-tier subsystem is very efficient and preserves the simple programming model. While this dissertation explores the essential parts of tiered transaction storage systems that leverage flash storage in a cost-effective way, several important questions are not fully addressed.

In this section, we discuss several interesting topics that can be pursued immediately and the aspirational topics that need more radical design changes and thus present future research opportunities.

7.2.1 Multi-Version Concurrency Control (MVCC)

Multi-Version Concurrency Control (MVCC) [73, 74] is a concurrency control mechanism for a highly concurrent transactional context. MVCC keeps multiple physical versions of a logical object in a database system. When a read-only transaction accesses an object while the object is being updated by another transaction, the system keeps the older version of the object and serves the read-only transaction without aborting. This technique trades off the capacity for the performance because multiple versions of an object consume additional storage space. Therefore, systems usually keep only a certain number of older objects and perform garbage collection to remove unnecessary older objects.

While current T2 does not support MVCC, it can additionally improve the overall performance for transactions by reducing the avoidable aborts resulting from high read latency.

The flash-tier region instance currently manages an object in a log-structured manner and updates the physical location in the mapping table entry atomically via CAS operations. Therefore, when an object is updated, the older versions of the object still physically exist until they are garbage-collected. Therefore, this characteristic simplifies the MVCC implementation for the flash-tier regions. To make the older versions accessible, we need to allow the mapping table to keep the older versions' locations, such as using a linked list, and to remove the older objects based on an MVCC policy.

7.2.2 Cache For Flash Tier

One preference of the design of T2 is to provide more memory than high-performance flash storage by utilizing some memory for cache. As a result, caching is not considered a way to improve performance. T2 implementation provides high performance by utilizing asynchronous and concurrent operations using flush buffers; however, the current design cannot hide the read latency from the flash storage at all.

Can cache help to reduce the read latency while sacrificing some of the memory space? As a tiered transaction storage system, this configuration introduces a new trade-off context for how to use DRAM. How much DRAM can be used for cache for the flash tier to improve overall performance? When DRAM is used as the memory tier, it can be accessed efficiently from remote servers and reduce read latency. In contrast, when DRAM is used as a cache for the flash tier, the read latency from the local flash tier cache can dramatically decrease. However, some objects which could be allocated in the memory tier should be migrated in the flash tier.

Moreover, even if cache for the flush tier improves the overall performance by consuming some capacity of the memory tier, effective use of cache is not trivial in the modern datacenter. The cache hit ratio generally depends on target workloads, but in the datacenter multiple tenants have different locality for their applications. Common cache policies, such as LRU and LFU, may improve the overall hit ratio. However, this does not imply

performance improvement for each individual application because each application's own locality can be easily discarded by another application and the system can be abused [75].

One plausible approach is to use a global cache that can serve all the flash-tier regions to improve utilization, similar to cache sharing and partitioning techniques in the computer architecture community. Having each tenant's id as an additional context during the allocation, the system can then track each tenant's utility and partition the cache size based on its usages [76].

7.2.3 Storage-class Memory

Recently, industry has started to adopt new technology, such as storage-class memory, and the performance and price of class storage memory are expected to be between DRAM and flash storage [56]. Although the two main suggestions are to use it as a slower but cheaper DRAM or as a faster but more expensive SSD, there are no standardized techniques that can be applied in datacenters. The question then becomes: how should we build the hierarchy of three different types of *storage* or *memory*? Which option will perform better for the same cost, using class storage memory as larger and cheaper DRAM or as faster and more expensive flash?

With class storage memory, we may need to reconsider our object model decision. We took an explicit approach to allow application developers to decide the residence of an object during the allocation, preferring the expected performance guarantee. However, the performance degradation to access an object in the storage class memory may be tolerable. In this case, implicit data migration can be achievable without causing negative performance impact. The datacenter workload changes over time, and we can migrate an object from *hot* DRAM to *warm* in class storage memory and to *cold* SSDs in a controlled and expected way.

REFERENCES

[1] J. C. McCallum, *Memory prices (1957-2017)*, `http://jcmit.net/mem2015.htm`.

[2] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13, Berkeley, CA, USA: USENIX Association, 2013, pp. 385–398.

[3] H. Zhang, G. Chen, B. C. Ooi, K. L. Tan, and M. Zhang, "In-memory big data management and processing: a survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 7, pp. 1920–1948, 2015.

[4] K.-L. Tan, Q. Cai, B. C. Ooi, W.-F. Wong, C. Yao, and H. Zhang, "In-memory databases: challenges and opportunities from software and hardware perspectives," *SIGMOD Record*, vol. 44, pp. 35–40, 2015.

[5] G. Graefe, H. Volos, H. Kimura, H. Kuno, J. Tucek, M. Lillibridge, and A. Veitch, "In-memory performance for big data," *Proc. VLDB Endow.*, vol. 8, no. 1, pp. 37–48, Sep. 2014.

[6] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: reliable, memory speed storage for cluster computing frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14, New York, NY, USA: ACM, 2014, 6:1–6:15, ISBN: 978-1-4503-3252-1.

[7] S. M. Rumble, A. Kejriwal, and J. Ousterhout, "Log-structured memory for dram-based storage," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, ser. FAST'14, Santa Clara, CA: USENIX Association, 2014, pp. 1–16, ISBN: 978-1-931971-08-9.

[8] S. M. Rumble, "Memory and object management in ramcloud," PhD thesis, Stanford University, 2014.

[9] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "Farm: fast remote memory," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14, Seattle, WA: USENIX Association, 2014, pp. 401–414, ISBN: 978-1-931971-09-6.

[10] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in ramcloud," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11, Cascais, Portugal: ACM, 2011, pp. 29–41, ISBN: 978-1-4503-0977-6.

[11] J. Gantz and D. Reinsel, "The digital universe in 2020: big data, bigger digital shadows, and biggest growth in the far east," *White Paper, IDC*, 2012.

[12] R. L. Villars, C. W. Olofson, and M. Eastwood, "Big data: what it is and why you should care," *White Paper, IDC*, p. 14, 2011.

[13] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile dram," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12, Washington, DC, USA: IEEE Computer Society, 2012, pp. 37–48, ISBN: 978-1-4503-1642-2.

[14] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: an introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[15] J. J. Levandoski, P.-A. Larson, and R. Stoica, "Identifying hot and cold data in main-memory databases," *2013 29th IEEE International Conference on Data Engineering (ICDE 2013)*, vol. 00, pp. 26–37, 2013.

[16] J. R. Douceur and W. J. Bolosky, "A large-scale study of file-system contents," in *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '99, New York, NY, USA: ACM, 1999, pp. 59–70, ISBN: 1-58113-083-X.

[17] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Commun. ACM*, vol. 60, no. 4, pp. 48–54, Mar. 2017.

[18] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No compromises: distributed transactions with consistency, availability, and performance," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15, Monterey, California: ACM, 2015, pp. 54–70, ISBN: 978-1-4503-3834-9.

[19] A. Leventhal, "Flash storage memory," *Commun. ACM*, vol. 51, no. 7, pp. 47–51, Jul. 2008.

[20] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th*

Annual International Symposium on Computer Architecture, ser. ISCA '09, New York, NY, USA: ACM, 2009, pp. 24–33, ISBN: 978-1-60558-526-0.

[21] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, Jun. 1970.

[22] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992.

[23] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '09, Seattle, WA, USA: ACM, 2009, pp. 181–192, ISBN: 978-1-60558-511-6.

[24] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to ssds: analysis of tradeoffs," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09, Nuremberg, Germany: ACM, 2009, pp. 145–158, ISBN: 978-1-60558-482-9.

[25] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "Tao: facebook's distributed data store for the social graph," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13, San Jose, CA: USENIX Association, 2013, pp. 49–60.

[26] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The ramcloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, 7:1–7:55, Aug. 2015.

[27] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The case for ramclouds: scalable high-performance storage entirely in dram," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 92–105, Jan. 2010.

[28] H. Garcia-Molina and K. Salem, "Main memory database systems: an overview," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 6, pp. 509–516, 1992.

[29] *RDMA*, `https://en.wikipedia.org/wiki/Remote_direct_memory_access`.

[30] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07, Stevenson, Washington, USA: ACM, 2007, pp. 205–220, ISBN: 978-1-59593-591-5.

[31] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10, New York, NY, USA: ACM, 2010, pp. 143–154, ISBN: 978-1-4503-0036-0.

[32] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: a fast array of wimpy nodes," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09, New York, NY, USA: ACM, 2009, pp. 1–14, ISBN: 978-1-60558-752-3.

[33] R. Cattell, "Scalable sql and nosql data stores," *SIGMOD Rec.*, vol. 39, no. 4, pp. 12–27, May 2011.

[34] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '12, New York, NY, USA: ACM, 2012, pp. 53–64, ISBN: 978-1-4503-1097-0.

[35] *Apache Hadoop*, `http://hadoop.apache.org/`.

[36] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03, Bolton Landing, NY, USA: ACM, 2003, pp. 29–43, ISBN: 1-58113-757-5.

[37] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: providing scalable, highly available storage for interactive services," in *Proceedings of the Conference on Innovative Data system Research (CIDR)*, 2011, pp. 223–234.

[38] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: google’s globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, 8:1–8:22, Aug. 2013.

[39] M. Ryu and U. Ramachandran, "Flashstream: a multi-tiered storage architecture for adaptive http streaming," in *Proceedings of the 21st ACM International Conference*

on Multimedia, ser. MM '13, Barcelona, Spain: ACM, 2013, pp. 313–322, ISBN: 978-1-4503-2404-5.

[40] J. Levandoski, D. Lomet, and S. Sengupta, "Llama: a cache/storage subsystem for modern hardware," *Proc. VLDB Endow.*, vol. 6, no. 10, pp. 877–888, Aug. 2013.

[41] J. Levandoski, D. Lomet, and K. K. Zhao, "Deuteronomy: transaction support for cloud data," in *Conference on Innovative Data Systems Research (CIDR)*, www.crdrdb.org, 2011.

[42] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance," in *USENIX 2008 Annual Technical Conference*, ser. ATC'08, Boston, Massachusetts: USENIX Association, 2008, pp. 57–70.

[43] M. Jung and M. Kandemir, "Revisiting widely held ssd expectations and rethinking system-level implications," in *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '13, Pittsburgh, PA, USA: ACM, 2013, pp. 203–216, ISBN: 978-1-4503-1900-3.

[44] B. Debnath, S. Sengupta, and J. Li, "Skimpystash: ram space skimpy key-value store on flash-based storage," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11, New York, NY, USA: ACM, 2011, pp. 25–36, ISBN: 978-1-4503-0661-4.

[45] *DRAM Average Unit Price - Statista*, `https://www.statista.com/statistics/298821/dram-average-unit-price/`.

[46] *redis*, `https://redis.io/`.

[47] *Apache Spark*, `http://spark.apache.org/`.

[48] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue, "Flat datacenter storage," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12, Berkeley, CA, USA: USENIX Association, 2012, pp. 1–15, ISBN: 978-1-931971-96-6.

[49] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik, "Anti-caching: a new approach to database management system architecture," *PVLDB*, vol. 6, pp. 1942–1953, 2013.

[50] L. Ma, J. Arulraj, S. Zhao, A. Pavlo, S. R. Dulloor, M. J. Giardino, J. Parkhurst, J. L. Gardner, K. Doshi, and S. Zdonik, "Larger-than-memory data management on modern storage hardware for in-memory oltp database systems," in *Proceedings of the 12th International Workshop on Data Management on New Hardware*, ser.

DaMoN '16, San Francisco, California: ACM, 2016, 9:1–9:7, ISBN: 978-1-4503-4319-0.

[51] .

[52] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min, "X-ftl: transactional ftl for sqlite databases," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13, New York, New York, USA: ACM, 2013, pp. 97–108, ISBN: 978-1-4503-2037-5.

[53] L. A. Barroso, J. Dean, and U. Hölzle, "Web search for a planet: the google cluster architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22–28, Mar. 2003.

[54] *Open cloudserver v2 (october 2014)*, `http://www.opencompute.org/wiki/Server/SpecsAndDesigns`.

[55] *UserBentocchmark*, `https://ssd.userbenchmark.com/`.

[56] *Intel Optan Memory - 3D Xpoint*, `http://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html`.

[57] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14, Amsterdam, The Netherlands: ACM, 2014, 15:1–15:15, ISBN: 978-1-4503-2704-6.

[58] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15, Melbourne, Victoria, Australia: ACM, 2015, pp. 707–722, ISBN: 978-1-4503-2758-9.

[59] *Persistent Memory File System*, `https://nvdimm.wiki.kernel.org/`.

[60] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09, Big Sky, Montana, USA: ACM, 2009, pp. 133–146, ISBN: 978-1-60558-752-3.

[61] *The serial ata international organization*, `https://sata-io.org/`.

[62] *Nvm express revision 1.3*, `http://nvmexpress.org/wp-content/uploads/NVM_Express_Revision_1.3.pdf`.

[63] *read vs. mmap (or io vs. page faults*, https://lists.freebsd.org/pipermail/freebsd-questions/2004-June/050371.html.

[64] *Windows file management functions*, https://msdn.microsoft.com/en-us/library/windows/desktop/aa364232(v=vs.85).aspx.

[65] S. Mittal and J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1537–1550, 2016.

[66] *Performance_information*, https://docs.microsoft.com/en-us/windows/desktop/api/psapi/ns-psapi-_performance_information.

[67] *SLA for Azure Cosmos DB*, https://azure.microsoft.com/en-us/support/legal/sla/cosmos-db/v1_0/.

[68] *Interlockedcompareexchange128 function*, https://msdn.microsoft.com/en-us/library/windows/desktop/hh972640(v=vs.85).aspx.

[69] *CreateFileA function*, https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea.

[70] J. Dean and S. Ghemawat, "Mapreduce: a flexible data processing tool," *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[71] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[72] D. Hilley and U. Ramachandran, "Persistent temporal streams," in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, ser. Middleware 09, Berlin, Heidelberg: Springer-Verlag, 2009.

[73] D. P. Reed, "Implementing atomic actions on decentralized data," *ACM Trans. Comput. Syst.*, vol. 1, no. 1, pp. 3–23, Feb. 1983.

[74] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185–221, Jun. 1981.

[75] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica, "Fairride: near-optimal, fair cache sharing," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA: USENIX Association, 2016, pp. 393–406.

[76] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings*

of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO 39, IEEE Computer Society, 2006, pp. 423–432, ISBN: 0-7695-2732-9.